creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사학위논문

대용량 폭증 데이터 처리를 위한 다중 메모리 서버 관리 기법

A Multiple Memory Server Management Scheme for Massive Explosive Data Streams

2015 년 8 월

서울대학교 대학원

전기 · 컴퓨터 공학부

장 준 혁

공학박사학위논문

대용량 폭증 데이터 처리를 위한 다중 메모리 서버 관리 기법

A Multiple Memory Server Management Scheme for Massive Explosive Data Streams

2015 년 8 월

서울대학교 대학원

전기 · 컴퓨터 공학부

장 준 혁

대용량 폭증 데이터 처리를 위한 다중 메모리 서버 관리 기법

A Multiple Memory Server Management Scheme for Massive Explosive Data Streams

지도교수 박 근 수

이 논문을 공학박사학위논문으로 제출함

2015 년 5 월

서울대학교 대학원

전기 · 컴퓨터 공학부

장 준 혁

장준혁의 박사학위논문을 인준함

2015 년 6 월

위 원 장 _____ (인)
부위원장 _____ (인)
위 원 _____ (인)
위 원 _____ (인)
위 원 _____ (인)

요약

메모리 가격이 큰 폭으로 하락하면서 대용량 데이터를 디스크에 저장하지 않고 다수의 서버 메모리에 모든 데이터를 저장하고 관리하는 메모리 기반 시스템이 여러 분야에서 활용되고 있다. 메모리 기반 시스템은 데이터의 전송량이나 처리량이 많은 증권사의 실시간 트레이딩 서버, 하둡 기반의 빅데이터 분석 서버 및 엔터프라이즈 서버 등에 적용되어 기존의 하드디스크 기반 분산 시스템 보다 더 우수한 성능을 보여주고 있다.

이러한 메모리 기반 시스템들은 메인 메모리에 저장된 데이터를 효율적으로 관리해야만 서비스의 품질과 시스템의 신뢰성을 높일 수가 있다. 그러나 기존의 메모리 데이터 관리 기법들은 사용자 요청의 특성이 달라지거나 결함이 자주 발생할 경우 시스템의 성능이 크게 떨어지는 문제점이 있다.

본 논문에서는 폭증 대용량 다중 서버 메모리 기반 시스템에서 데이터 관리를 위한 효율적인 해시 기법과 복구 기법을 제안한다. 제안한 해시 기법은 데이터 검색 시에는 이중화된 해시 테이블을 사용하고 데이터 삽입 시에는 연결형 해시 테이블을 사용하여 데이터 검색과 삽입이 모두 상수 시간에 수행되게 한다. 또한 제안한 복구 기법은 데이터 서버 결함 발생 시 다수의 백업 서버에서 동시에 데이터 서버로 백업 데이터를 전송할 수 있도록 하여 사용자의 요청 처리 시간을 줄인다.

본 논문에서는 제안한 해시 기법과 복구 기법의 성능 분석을 위해 시간 복잡도를 확률적으로 분석하여 검색과 삽입이 평균적으로 상수 시간에 수행됨을 확인하고, 사용자 요청에 대한 예상 처리 시간을 모델링 기반의 정형화된 수식으로 유도하여 제안한 기법의 예상 처리 시간이 줄어듬을

보인다. 또한 성능 평가를 위해 제안한 기법을 구현하고 실제 데이터 워크로드를 기반으로 실험하여 기존 다중 서버 메모리 기반 시스템의 관리 기법보다 제안한 기법이 데이터 처리율이 높아지고 복구 시간이 짧아짐을 보인다.

주요어: 메모리 기반 시스템, 데이터 관리, 폭증 데이터, 해시 테이블, 데이터 복구, 성능 분석

학번: 2010-20882

목차

그림 목차

표 목차

제 1 장 서론

1.1 연구 배경

네트워크 기술이 발전하고 메모리 기반 시스템에 대한 기술적 연구가 활발히 이루어짐에 따라 활용 범위 또한 넓어지고 있다[1]. 메모리 기반 시스템은 다양한 계층의 기술이 접목되어 서로 연동되면서 서비스를 제공하게 되는데, 최근에는 웹서비스, 금융, 스토리지 서비스 등 다양한 분야의 응용에서 사용되고 있다. 이에 따라 응답 시간, 처리율, 결함허용, 확장성 등에 있어서 서로 다른 수준의 요구사항을 포함하게 되었으며 워크로드의 특성 또한 어떤 응용을 사용하는가에 따라 달라지게 되었다[2]. 따라서 사용되는 응용의 특성과 그 요구사항을 고려하여 데이터를 적재하고 관리할 수 있어야 한다.

예를 들어 HFT(High Frequency Trading)와 같은 분야에서 이런 특징이 두드러지는데, HFT 분야에서 처리되는 금융 거래 데이터는 마이크로초 단위의 처리 시간을 보장해야 하며 때때로 특정 시간에 트래픽이 급증하는 패턴을 보인다[3]. 이러한 응용은 데이터의 정합성을 보장하는 한편 확장성, 결함허용 기능 등의 수준을 떨어뜨리더라도 응답 시간을 줄이는 데 주안점을 두게 된다.

이러한 환경의 변화에 따라 다중 서버 메모리 기반 시스템에서 데이터 관리의 중요성 또한 높아지게 되었다. 특히 급증하는 데이터를 빠르게 처리하기에는 기존의 디스크 기반 시스템으로는 한계가 있다는 문제점이 대두됨에 따라 메모리 기반 시스템의 사용이 증가하는 추세이다. 메모리 기반 시스템은 메모리의 다소 비싼 가격으로 인해 수요가 많지 않았으나, 메모리의 가격이 하락하고 더 높은 처리 속도가 요구됨에 따라 수요가 증가하고 있다[4]. 특히 많은 데이터가 발생하는 증권사의 실시간 금융 트레이딩, 온라인 게임, 모바일 어플리케이션, 통신사의 세션관리, 소셜 미디어 등 다양한 분야에서 효과적이라고 알려져 있다[7].

데이터의 주 저장 공간의 변화는 과거보다 더욱 더 빠른 데이터 처리가 가능

해지게 하였다. 디스크 기반 시스템은 대용량 데이터를 저장하기에는 용이하나 대용량 데이터를 고속으로 처리하기에는 적합하지 않다. 반면 메모리 기반 시스템은 디스크 입출력으로 인한 지연시간 없이 데이터를 처리하므로 디스크 기반 시스템보다 빠른 데이터 처리가 가능하다. 다만 메모리 기반 시스템은 메모리의 크기에 따라 저장 공간이 제한된다는 문제점이 존재하는데, 최근 메모리 가격 하락에 따라 서버 시스템들의 메모리 크기가 증가하고 있어 메모리 크기 제한으로 인한 문제는 완화되는 추세이다. 또한 64비트 아키텍처 가운데 AMD64(x86-64) 아키텍처는 가상 주소에 48비트를 사용하여 최대 256 테라바이트가 사용 가능하며, IA-64 아키텍처는 64비트를 모두 사용한다고 알려져 있다. 이 가운데 사용자 모드에서 사용 가능한 가상 주소 공간은 일반적으로 8TB이다[5][6].

제한된 저장 공간을 갖는 메모리 기반 시스템에 많은 양의 데이터를 효율적으로 저장하기 위해서는 기존의 복잡한 형태의 관계형 데이터베이스 시스템 보다는 간단한 형태의 비정형 데이터베이스 시스템이 적합하다. 비정형 데이터 모델인 NoSQL은 “Not Only SQL” 혹은 “Not Relational”을 의미하며, 기존 관계형 데이터베이스 시스템보다 좋은 수평 확장성을 갖기 때문에 빠르고 효율적인 메모리 기반 시스템 구축에 적합하다[8].

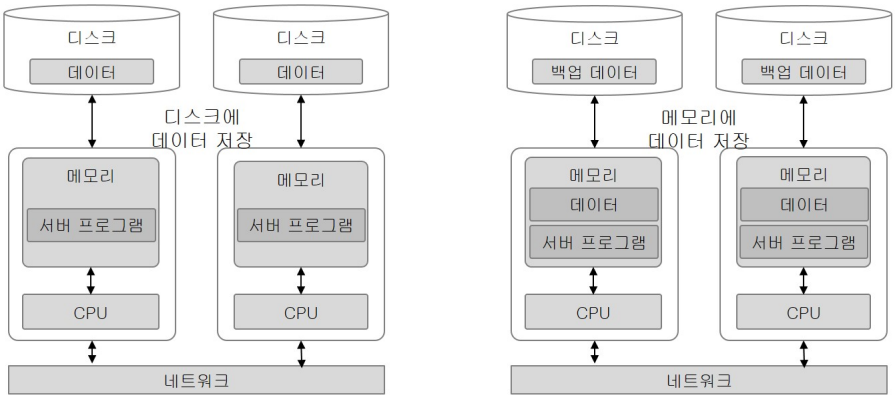

그림 1.1 메모리 기반 시스템

1.2 연구 목적 및 범위

메모리 기반 시스템에서 데이터는 다수의 서버에 적절히 분배되어야 한다. 이 때 워크로드의 읽기/쓰기 빈도, 데이터 크기 등은 응용에 따라 다양한 분포를 가질 수 있다. 서버 메모리에 데이터를 효율적으로 저장 및 관리하기 위해서는 제한된 메모리 공간을 효율적으로 사용해야 하며 사용자의 요청은 가능한 한 빠르게 처리되어야 한다. 또한 데이터의 사본을 유지 관리하여 장애 발생 시 빠르게 복구될 수 있어야 한다[9].

Facebook 등의 연구 결과에 따르면 메모리 기반 시스템에 적재되는 데이터는 몇 가지 독특한 특징을 가진다[10][11]. 첫째, 각 데이터의 크기가 수십 바이트 수준으로 매우 작은 반면 데이터의 수는 매우 많다. 따라서 전체 데이터 크기에서 실제 데이터를 뺀 메타 데이터의 비율이 크게 높아지므로 데이터 관리 기법의 설계에 따라 공간 효율성이 크게 좌우된다는 특징이 있다. 둘째, 데이터의 구조가 단순하다. 기존의 데이터 관리 기법은 각 데이터가 긴밀하게 연결되어 있으며 데이터 관리 시스템이 복잡한 요청을 지원해야 하였으나, 메모리 기반 시스템의 데이터는 대부분이 개별 데이터에 대한 접근으로 이루어져 있으며, 데이터 사이에 관계가 있더라도 비교적 단순한 편이므로 구조가 복잡하지 않고 속도가 빠른 관리 기법이 사용된다. 셋째, 애플리케이션에 따라 다양한 워크로드 특성이 요구된다. 읽기와 쓰기 요청 중 읽기 요청이 대부분을 차지한다는 연구 결과가 있었으나, 최근의 추세에 따르면 메모리 기반 시스템의 활용 범위가 넓어짐에 따라 애플리케이션의 다양성만큼 워크로드의 특성 또한 다양해지고 있다. 기존의 메모리 데이터 관리 기법은 이와 같은 연구 결과에 적합하게 설계되어 있으므로 최근의 경향에는 맞지 않다. 메모리 기반 시스템의 적용 범위가 웹서비스, 소셜 네트워크, 금융 거래 시스템 등 다양한 분야로 확장됨에 따라 요구되는 특성 또한 다양해지고 있기 때문이다.

특히 본 논문에서 대상으로 하고 있는 실시간 금융 시스템은 다수의 데이터 서버 메모리에 데이터를 분산 적재하며, 데이터 서버 별로 백업 서버를 두어 데이터를 백업한다. 실시간 금융 시스템의 특성상 데이터의 검색, 삽입 성능과 접근

지연 시간 단축이 가장 중요한 요구 사항이 된다. 금융 거래의 특성상 워크로드는 읽기에 편중되어 있지 않으며 특정 시간대에 데이터 접근 요청이 폭증하는 특징을 가진다[12][13]. 따라서 읽기/쓰기 요청의 비율이 다양하며, 시간대에 따라 사용자 요청이 크게 증가하는 형태의 워크로드에 적합한 데이터 관리 기법의 연구가 필요하다.

메모리 데이터 관리에서 데이터의 빠른 처리를 위해 데이터를 저장 및 관리하는 핵심적인 부분은 해시 테이블로 구현되는데, 주로 체인 해시 기법 등이 사용되며 최근 쿠쿠 해시 기법 등이 제안되었다[14][16]. 하지만 응용의 워크로드가 다양해지는 데 비해 기존의 메인 메모리 데이터 관리 기법은 읽기 요청의 처리 성능을 높이는 데 중점을 두고 있어 다양한 요구사항을 충족시키지 못하고 있다[10]. 따라서 본 논문에서는 읽기 중심의 워크로드 뿐만 아니라 쓰기 중심의 워크로드에서도 빠른 처리 성능을 보여줄 수 있는 메모리 데이터 관리 기법을 제안한다.

기존의 메모리 기반 시스템은 예기치 않은 시스템 결함으로 인한 데이터 손실 방지를 위해 다양한 결함허용 기법들을 사용하는데, 동일한 시스템에 데이터를 기록하는 체크포인팅 기법 및 로깅 기법, 다른 시스템에 데이터를 기록하는 이중화 기법 등을 사용하여 디스크에 백업 데이터를 기록하고 장애 발생 시 데이터를 복구한다. 하지만 이와 같은 디스크 기반의 결함허용 기법은 시스템 장애 발생 시 빠른 복구 속도를 제공하지 못한다. 따라서 본 논문에서는 메모리 기반 복구 기법을 제안한다. 제안 기법은 데이터 서버에 저장된 데이터의 사본을 다수 백업 서버의 메인 메모리에 분산하여 저장하며, 따라서 시스템 장애 발생 시 다수의 백업 서버에서 동시에 데이터를 전송하여 복구 시간을 단축시킬 수 있다.

제안 기법은 메모리 데이터 관리에 사용되는 기존 해시 기법에서 삽입 요청의 응답 시간이 느려지는 문제점을 보완하여 다양한 패턴의 부하에 대해 처리 성능을 높이고, 백업 및 복구를 통해 안정성을 높이는 데 목적을 두고 시간 복잡도와 예상 처리 시간 등을 분석한다. 또한 제안 기법을 LINUX 운영체제에서 구현하고, 실제 기업에서 사용하고 있는 시스템 모델 및 워크로드를 적용하여 성능을 확인한다.

1.3 논문의 구성

본문의 구성은 다음과 같다. 2장에서는 관련 연구로 기존의 메모리 데이터 관리 기법들에 대해 기술하고, 메모리 접근 지연 시간을 줄이기 위한 해시 알고리즘들에 대해 설명한다. 3장에서는 시스템 모델을 제시하고, 4장에서는 다양한 워크로드에 대해 빠르고 안정적으로 메모리 데이터를 처리할 수 있는 해시 기법을 제안한다. 5장에서는 제안 기법의 성능을 모델링한다. 6장에서는 메모리 기반 시스템의 결함허용 기법에 대해 기술하고 제안 기법의 예상 처리 시간을 수학적으로 분석한다. 7장에서는 제안 기법의 구현 결과를 보이고 워크로드의 변화에 따른 제안 기법의 성능을 확인한다. 마지막으로 8장에서는 결론을 맺고, 제안 기법과 관련한 향후 연구에 대해 설명한다.

제 2 장 관련 연구

본 장에서는 본 본문과 관련된 기존 연구들을 살펴본다. 먼저 2.1 절에서는 메모리 데이터 관리의 필요성에 대해서 설명한다. 2.2 절에서는 메모리 데이터를 관리하는 기존의 기법들에 대해 설명한다.

2.1 메인 메모리 데이터 관리

본 절에서는 디스크에 저장된 데이터를 보다 빨리 접근하고 처리하기 위해 개발된 메모리 데이터 관리 기법에 대해 설명한다. 대규모 데이터를 처리하는 응용에서 디스크에 데이터를 저장하지 않고 메인 메모리에 저장하여 데이터의 접근 속도를 향상시키는 연구가 진행되어 왔다. 디스크에 대규모 데이터를 저장하지 않고 메모리에 데이터를 저장함으로써 디스크에 저장된 데이터를 읽고 쓰는 것에 비해 처리 시간을 상당히 줄일 수 있다[1]. 표 2.1은 디스크와 메모리 간의 접근 지연 시간과 읽기 시간을 보여 준다.

표 2.1 디스크와 메모리 사이의 접근 시간 비교

Action	Time (ns)
Main memory access	100
Read 1 MB sequentially from memory	250,000
Disk seek	5,000,000
Read 1 MB sequentially from disk	30,000,000

응용의 요구 사항이 복잡해지고 더 높은 성능이 요구됨에 따라 메모리에 데이터를 저장할 필요성이 증가하게 되었다. 메모리 데이터 관리 기법의 연구는 데이터 접근 속도를 향상시키는 수준에 머무르지 않고 다중 코어 CPU의 사용으로 효율적으로 병렬 처리하는 연구와 CPU와 메모리 사이의 속도 차이를 고려하여 데이터를 효율적으로 분산 메모리에 저장하고 이를 관리하여 시스템 전체 성능을 극대화시키는 등의 연구가 활발하게 진행되고 있다.

메모리 데이터 관리에 관한 연구로는 램클라우드(RAMCloud)가 대표적이다. 램클라우드는 기존의 디스크 기반 데이터 관리 기법의 문제점을 해결하기 위해 제안되었다. 램클라우드는 기존의 디스크 기반 서버와 달리 데이터를 디스크가 아닌 다중 서버 메모리에 분산 적재하도록 하여 시스템의 확장성을 높이고 접근 지연 시간을 단축시키도록 한다. 또한 메모리의 데이터 로그를 비동기적으로 디스크에 백업하고, 복구 시에 다중 서버의 디스크에서 로그를 읽어 복구 시간을 단축시키도록 하였다. 초기의 램클라우드는 아이디어 수준에 머물렀으나, 현재에는 이를 기반으로 다양한 연구가 진행되고 있으며 2장에서 소개하는 데이터 관리 기법들 중 다수가 램클라우드의 영향을 크게 받았다고 볼 수 있다[17][18].

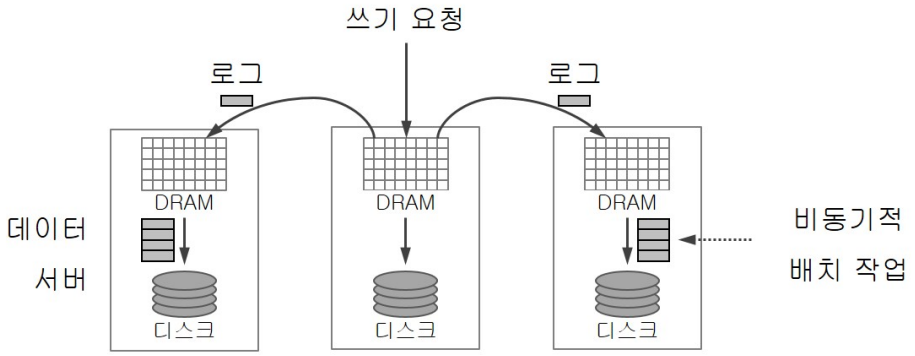

그림 2.1 램클라우드의 데이터 저장 기법

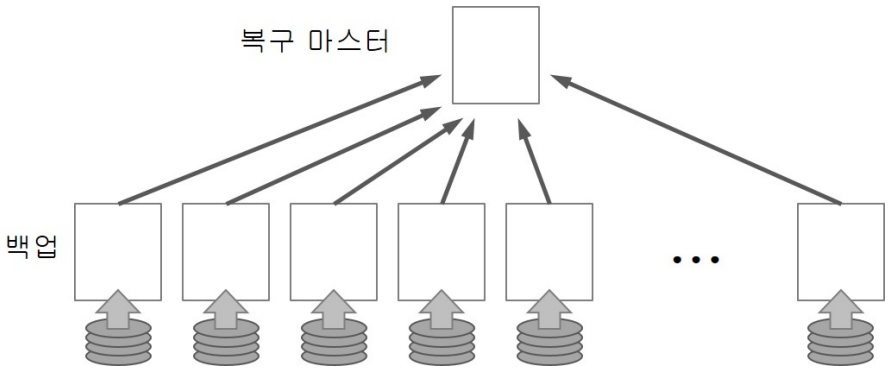

그림 2.2 램클라우드의 데이터 복구 기법

특히 실시간으로 폭증 데이터를 처리해야 하는 메모리 기반 시스템에서는

메모리 데이터의 접근과 변경이 빈번하게 이루어지기 때문에 메모리의 데이터를 효율적으로 관리하는 기법의 중요성이 더욱 증대되고 있다[19][20]. 메모리에 저장된 데이터를 효율적으로 관리하는 기법으로는 칼럼 기반 저장시스템과 결합된 인-메모리 데이터 관리 기법[21], 다중 코어 및 시스템 간 사이의 병렬 처리 기법[22] 그리고 압축 기법을 활용한 성능 향상 및 저장 공간 절약(Using Compression for Performance and to Save Space) 기법[23] 등이 있다.

먼저 칼럼 기반 저장 시스템과 결합된 메모리 데이터 관리 기법은 질의 처리 연산에 사용되는 필요한 칼럼만을 메모리에 저장함으로써 기존의 행 기반 관계형 데이터베이스 관리 시스템들에서 질의 결과와 관련 없는 모든 칼럼을 메모리에 저장하는 열 기반의 데이터 관리 시스템보다 데이터의 접근 속도를 증가시킨 기법이다. 이는 대규모 데이터를 처리하는 응용에서 모든 칼럼을 사용하지 않고 소수의 칼럼만을 사용하여 질의 처리 연산을 반복적으로 수행하는 경우가 많기 때문이다. 두 번째로 다중 코어 및 시스템 간의 병렬 처리 기법은 다중 코어 CPU를 활용하여 메모리에 저장된 데이터 처리를 여러 코어에 분산 시켜 병렬적으로 작업을 처리하여 데이터 접근 속도를 향상시킨 것이다. 이 기법은 메모리에 데이터를 저장하는데 있어 하나 또는 여러 코어에 각 칼럼을 할당하는 수직 단편화 (Vertical Fragmentation) 기법과 테이블의 행들을 여러 개의 그룹으로 나누고 분산된 CPU에 배정하는 수평 단편화 (Horizontal Fragmentation) 기법들로 나눌 수 있다. 이러한 단편화 기법은 효율적으로 대규모 데이터를 처리하는데 있어서 각 CPU가 처리할 데이터의 양을 줄여 성능을 향상시켰으나, 칼럼 간의 의존성이 큰 행 기반 관계형 데이터 관리시스템보다 칼럼 간의 의존성이 작은 칼럼 기반 데이터 관리 시스템에서에서만 유용하게 사용된다.

마지막으로 압축 기법을 활용한 성능 향상 및 저장 공간 절약 기법은 데이터 압축을 통해 메모리 저장 공간의 낭비를 줄이고 데이터 간의 관계를 중복을 최소화하여 성능을 향상시킨 기법이다. 이 기법은 칼럼 기반 데이터 저장 시스템에서 더 좋은 성능을 보인다. 이는 칼럼 기반 데이터 저장 시스템은 같은 형태의 데이터 타입을 가지고 있으며 많은 데이터들의 값이 동일하기 때문에 압축 효율을 높일

수 있으며 압축으로 인해 데이터의 밀집도가 높아지고 질의 처리 시에 사용되는 메모리 공간의 크기가 줄어들게 되기 때문이다. 그러나 행 기반 관계형 데이터 관리 시스템에서는 불필요한 데이터가 메모리에 로드 되어 있어 메모리가 많이 낭비되는 문제가 발생하기도 한다.

2.2 메모리 데이터 관리 모델

본 절에서는 메인 메모리 데이터를 관리하는 다양한 모델에 대해 설명한다. 메모리에 다양한 형태로 저장된 데이터를 관리하기 위해서 키-값(Key-Value), 도큐먼트(Document), 칼럼족(Column Family) 및 그래프 데이터 모델이 사용된다. 먼저 키-값 데이터 모델은 가장 단순한 형태로 키와 데이터 값을 사상하여 저장한다. 키는 검색에 필요한 인덱스의 형태로 존재하며 일반적으로 관계형 데이터베이스에서 사용하는 기본키와 외래키의 개념은 포함하지 않는다. 주로 해시 함수와 같이 빠른 검색과 확장성을 보장해주는 기법을 활용한다. 웹 사이트의 방문 기록, 사용자의 설정 값 등의 정보를 저장하기에 좋은 모델이다. 이 모델을 사용하는 시스템으로 Redis, Riak, 그리고 Tokyo Cabinet 등이 있다. Redis는 C로 프로그래밍 되어 있으며 데이터를 저장하는 서버와 데이터를 검색 및 갱신하는 클라이언트 라이브러리로 구성되어 있다. 클라이언트는 분산형 해시 테이블을 통해 원하는 데이터가 저장된 서버를 검색하며 서버는 데이터를 메모리에 저장하고 복사본을 디스크에 저장한다[14]. Riak은 분산형 시스템 구현에 적합한 Erlang으로 프로그래밍 되어 있으며 키-값 모델과 도큐먼트 모델을 모두 지원한다. 각 객체들은 JSON 형식으로 저장되며 각 객체는 기본키를 통해 접근이 가능하다. 데이터 분산처리를 위해 저장된 데이터를 분할(Partition)하는 샤딩(Sharding)을 지원한다[15]. Tokyo Cabinet은 C로 프로그래밍 되어 있으며 서버 측면의 cabinet과 클라이언트 측면의 tyrant로 구성되어 있다. 서버는 데이터에 대한 해시 인덱스, B-tree, 고정 크기 레코드 테이블, 가변 크기 레코드 테이블 등의 방식으로 메모리에 저장한다[8].

두 번째로, 도큐먼트 데이터 모델은 저장과 질의의 주체가 문서 단위로 이루

어진다. 문서는 PDF, 한글, Microsoft 워드 문서와 같은 텍스트 문서뿐만 아니라 XML (Extensible Markup Language), JSON (JavaScript Object Notation) 과 같은 문서도 포함한다. 도큐먼트 모델은 대량의 도서, 출판물과 같은 텍스트 정보를 포함한 데이터를 저장하기에 적합하며 관계형 테이블의 각 행에 대한 내용도 문서화하여 저장할 수 있다. 이러한 모델을 사용하는 시스템으로는 Simple, Couch, 그리고 Mongo 등이 있다. Simple은 Amazon 사에서 클라우드 서비스를 제공하기 위한 프레임워크인 Elastic Compute Cloud(E2C) 에 사용되는 데이터 저장 시스템으로 간단한 형태로 문서들을 저장한다. 문서는 그룹화된 도메인 (Domain) 으로 관리되고 한 문서는 여러 도메인에 포함될 수 있기 때문에 다중 인덱스 (Multiple Indexes) 를 지원한다. 도메인을 대상으로 SELECT 문을 활용함으로써 대규모의 검색이 효율적으로 이루어 질 수 있도록 해 준다. Couch 는 Erlang으로 프로그래밍 되어 있으며 도메인과 유사한 컬렉션 (Collection) 으로 문서를 그룹화 한다. 문서는 문서, 숫자 및 문서의 리스트로 구성되어 있으며 모든 컬렉션은 보조 인덱스 (Secondary Index) 를 포함해야 한다[26].

Mongo는 C++로 프로그래밍 되어 있으며 컬렉션 개념과 문서에 대한 질의 메커니즘을 제공한다. 사용자에 의해 수동으로 데이터 파티션을 진행하지 않고 서버에서 자동으로 샤딩 기법을 제공한다. 일반적으로 비절차형 데이터 관리 시스템에서 제공하지 못하는 원자적 데이터 관리를 지원한다[24].

세 번째로 칼럼족 데이터 모델은 그림 2.3과 같이 관계형 데이터베이스의 테이블과 유사하지만 행마다 각 칼럼의 이름과 개수가 다를 수 있다는 점이 다르다. 각 칼럼족은 "키"와 "값"의 쌍으로 구성되고 키를 통하여 행을 접근할 수 있다. 그러나 칼럼족 모델은 이벤트 모니터링, 콘텐트 관리 시스템 등에 활용되며 트랜잭션의 ACID 보장하지 못하고 질의가 복잡할 경우 성능이 떨어지는 문제가 발생한다. 이러한 칼럼족 모델을 사용하는 시스템으로는 Bigtable 그리고 HBase 등이 있다.

Bigtable은 Google 사에서 대규모 데이터 저리를 위해 개발된 시스템으로 펩타 단위의 데이터를 수천 개의 서버에 분산해서 저장할 수 있도록 해준다. 웹

기반 데이터의 효율적인 저장을 위해서 행 키(Row Key), 열 키(Column Key), 타임스탬프로 구성된 인덱스 맵을 제공한다. 이러한 칼럼족을 생성 및 삭제할 수 있는 API를 제공하며 C++ 객체를 통해 Bigtable의 데이터를 읽고 쓸 수 있다. 각 테이블들은 3계층의 B+-tree로 구성되어 루트는 각 문서들의 위치를 저장하는 메타데이터를 포함하고 있다 [27]. HBase는 Java로 프로그래밍 되어 있으며 Hadoop의 분산 파일 시스템으로 사용된다. 갱신된 내용은 메모리에 저장하며 주기적으로 디스크의 파일에 저장한다. 로그를 지속적으로 기록하여 시스템에서 충돌이나 오류 발생 시 이를 복구할 수 있는 메커니즘을 제공한다. 행에 대한 원자적 연산을 보장하며 B-tree 구조를 사용하여 빠른 검색과 정렬이 가능하다[21].

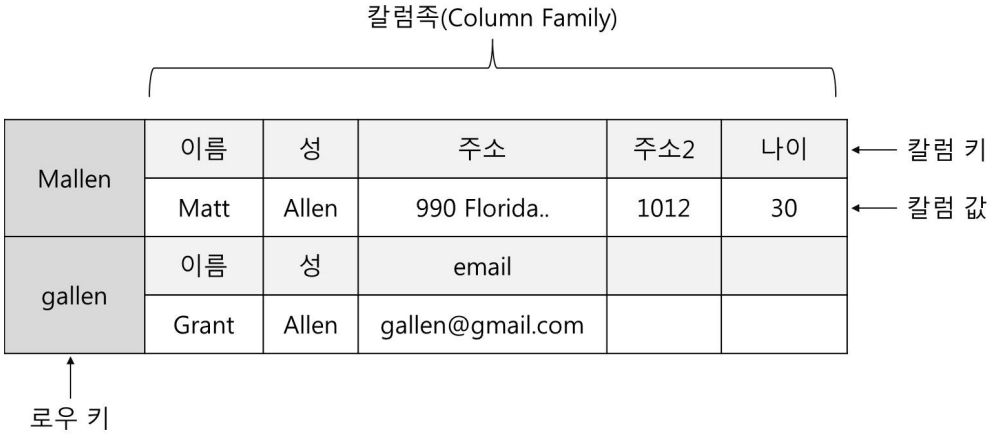

그림 2.3 칼럼족(Column Family) 모델

마지막으로 그래프 데이터 모델은 데이터의 관계에 초점을 맞추고 데이터와 데이터를 링크로 연결한다. 즉 데이터 포인트는 노드(Node)를 나타내고, 이들 사이의 관계는 에지(Edge)를 통해 연결된다. 이러한 그래프 모델은 Facebook, Twitter, 카카오톡과 같은 소셜네트워크서비스(Social Network Service, SNS) 관련 데이터를 저장하기에 적합하다. 그래프 모델을 사용하는 시스템으로는 Orient와 Neo4j 등이 있다.

Orient는 Document 모델의 유연성과 graph 모델의 표현력을 결합한 형태로

대규모의 데이터에 대해 빠른 연산 속도를 제공한다. 문서들 간의 관계성을 그래프의 형태로 표현하기 때문에 조인 연산이 필요 없어 데이터 검색 속도가 빠르고, 문서들 간의 중복도 최소화 하였다. 관계 탐색에 있어서 상수 시간의 오버헤드를 보장하며 분산 처리를 위한 샤딩을 지원하고 트랜잭션의 ACID까지 보장한다[28]. Neo4j는 Java로 프로그래밍 되어 있으며 graph를 사용하여 데이터를 표현하기 때문에 직관적이다. 데이터의 안정성을 위해서 트랜잭션의 ACID를 보장하며 수십 억 개의 노드, 관계, 속성의 조합이 가능하다. 편리한 샤딩을 통해 데이터를 분산된 서버에 저장할 수 있으며 그래프 추적을 통해 질의할 수 있다[29].

2.3 메모리 접근 시간을 줄이는 해시 기법

본 절에서는 메모리에 데이터를 저장하고 접근하기 위하여 사용하는 해시 함수 기법에 대해 설명한다.

2.3.1 체인 해시

일반적으로 해시 함수를 사용하여 데이터 저장 위치를 지정할 경우 항상 동일한 위치에 데이터가 저장되기 때문에 저장할 데이터의 크기가 해시 테이블의 크기보다 크다면 충돌 (Collision)이 발생 한다. 즉 서로 다른 *hash(x) mod N*과 *hash(y) mod N*의 결과가 동일할 경우 같은 공간에 데이터를 저장해야 하기 때문에 둘 중 하나의 값만 저장할 수 있다.

그러나 기존의 해시 구조에 연결 리스트 (Linked List)를 활용하여 충돌되는 데이터를 체인 (Chain)의 형태로 연결하면 기존의 해시 테이블의 구조를 변경하지 않고도 충돌 문제를 효과적으로 해결할 수 있다 [34]. 그림 2.4는 체인 해시 기법을 활용한 해시 테이블을 보여준다.

해시 테이블의 각 슬롯이 연결 리스트로 이루어져, 충돌 발생 시 노드를 추가하여 데이터가 삽입된다. 체인 해시 기법은 데이터 삽입이 상수 시간에 이루어지므로 효율적이지만, 리스트가 길어질수록 데이터 검색 시간이 늘어나 최악의 경우 검색 시간을 보장하지 못하는 단점이 있다. 이러한 체인 해시 기법은 Broder 등

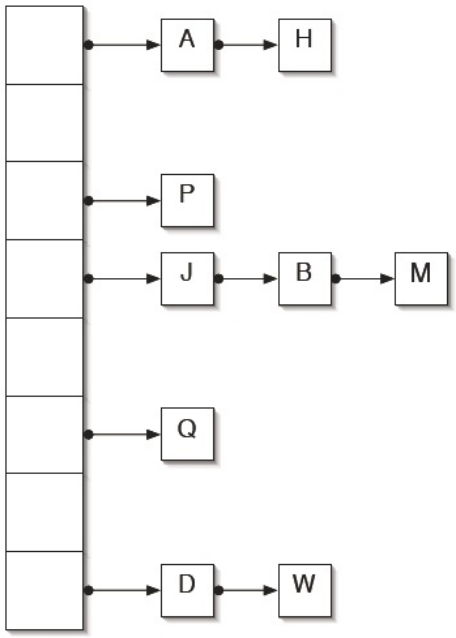

그림 2.4 체인 해시 테이블

[35] 이 제안한 웹페이지의 URL (Uniform Resource Locator) 저장, Bruna 등 [36] 이 고안한 삼차원 도형의 단면을 이차원으로 변환하는 메소드, Deodhar 등[37] 이 메모리 사용을 최소화 할 수 있는 응용 구현 등 다양한 분야에서 활용되고 있다[38][22][39].

2.3.2 양방향 해시

체인 해시 기법을 사용할 경우 해시 테이블 상의 충돌 문제를 해결할 수 있는 장점이 있지만 체인의 길이가 길어지면 필요한 데이터를 검색하는데 필요한 시간을 예측하기 힘들어지고 최악 검색 시간 (Worst-case Search Time) 이 길어진다. 검색 시간에 대한 예측 불가능 문제를 해결하기 위해서 양방향 해시 기법이 제안되었다 [33]. 그림 2.5는 양방향 해시 기법의 동작 원리를 보여준다.

해시 테이블에 크기가 n 이고 해시 테이블에 m 개의 데이터를 할당한다고 가정할 때 서로 다른 해시 값을 도출하는 *f(x)*와 *g(x)* 해시 함수를 정의한다. 만약

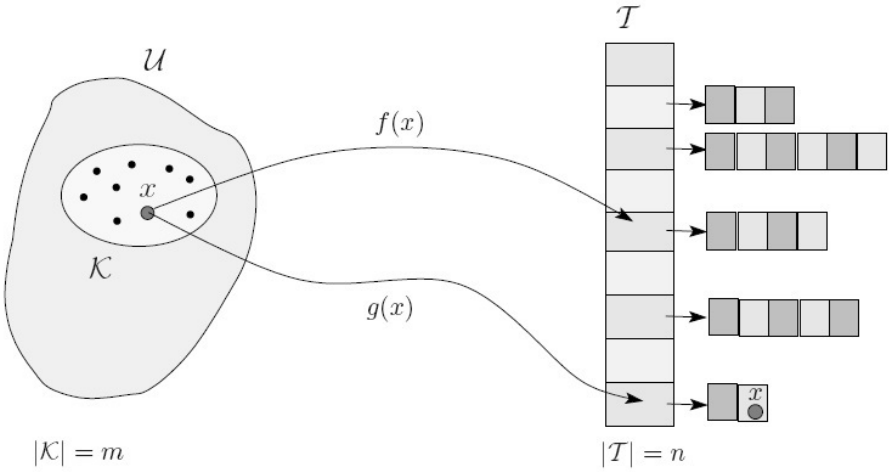

그림 2.5 양방향 해시 테이블

해당 아이템 x의 해시 값을 계산한 후에 충돌이 발생하는 경우 두 개의 테이블 슬롯 중에서 리스트의 길이가 짧은 곳에 새로운 값을 삽입한다.

2.3.3 쿠쿠 해시

쿠쿠 해시(Cuckoo Hash) 기법은 앞서 언급한 양방향 해시 기법을 사용하더라도 해시 테이블에 저장된 데이터를 검색하는데 상수 시간의 최악 검색 시간을 보장하기 어렵기 때문에 이를 해결하기 위한 대안으로 제안되었다 [47]. 상수 시간의 검색 시간을 보장하는 가장 간단한 방법은 해시 테이블 상에서 충돌이 발생하지 않게 만드는 방법으로 삽입할 데이터의 전체 개수와 동일한 개수의 해시 값을 도출하는 완전 해시 함수 (Perfect Hash Function)를 사용하는 것이다. 그러나 완전 해시 함수를 활용하는 기법의 경우 동적으로 데이터의 개수가 변하는 상황을 지원하지 못하고 해시 테이블의 활용률이 떨어지는 문제가 발생한다. 쿠쿠 해시 기법에서는 양방향 해시 기법에서와 마찬가지로 2개의 해시 함수와 2개의 해시 테이블 사용하는 해싱 기법이다. 하나 의 키는 2개의 해시 함수를 사용해 각 테이블마다 하나의 슬롯을 가진다. 키가 해시 테이블 내에 존재한다면 해당 키는 반드시 두 개의 슬롯 중 하나에 위치한다. 이를 통해 데이터를 저장할

두 곳의 슬롯을 지정한다. 여기에 최소한 한 곳은 데이터를 저장할 수 있도록 해준다는 가정을 하면 체이닝 기법에 구현에 필요한 연결 리스트 구조 없이 배열만으로 해시 테이블을 구성할 수 있다. 만약 새로운 데이터 x를 해시 테이블에 저장한다고 했을 때, 데이터를 저장할 두 슬롯 모두 이미 다른 데이터가 저장되어 있을 가능성이 있다. 이러한 상황을 해결하기 위해서 그림 2.6과 같이 기존에 저장되어 있던 데이터 y를 비어 있는 곳으로 옮기고 x를 y의 자리에 할당하는 방법이 있다.

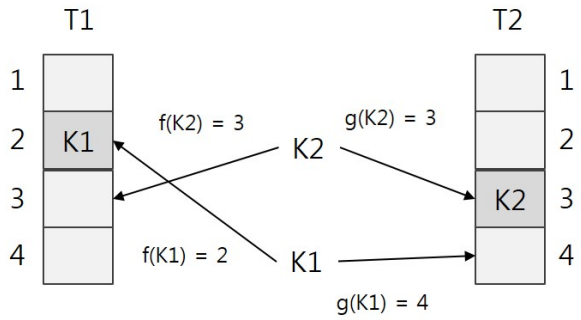

그림 2.6 쿠쿠 해시 테이블

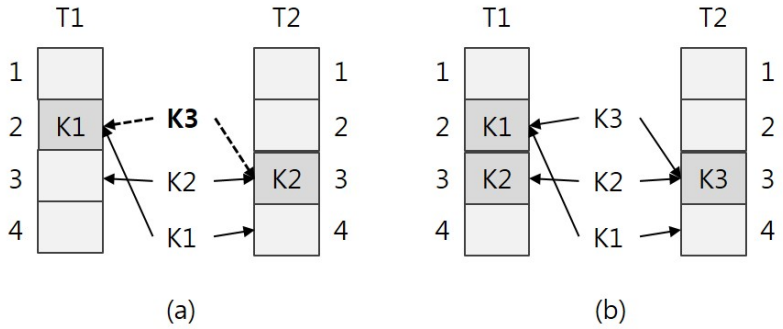

그림 2.7 쿠쿠 해시 테이블의 삽입

이러한 쿠쿠 해시 기법의 삽입 알고리즘은 그림 2.7과 같다. 데이터를 삽입할 때, 해시 함수를 통해 찾은 두 슬롯 중 빈 곳이 있다면 비어 있는 슬롯에 삽입한다. 두 슬롯 모두 다른 키가 저장되어 있다면, 두 슬롯 중 하나를 택해 기존의 키를 다른 곳으로 쫓아낸 후 빈 자리에 삽입한다. 각 키는 두 개의 슬롯 중 하나에

```
procedure insert(x)
  if T[h1(x)] = x or T[h2(x)] = x then return;
  pos ← h1(x);
  loop n times {
    if T[pos] = NULL then { T[pos] ← x; return};
    x ↔ T[pos];
    if pos= h1(x) then pos← h2(x) else pos← h1(x);}
  rehash(); insert(x)
end
```

그림 2.8 쿠쿠 해시의 삽입 알고리즘

위치하므로, 쫓겨난 키는 다른 테이블에 있는 자신의 슬롯으로 이동되며, 이 과정은 빈 슬롯을 찾을 때까지 반복된다.

일반적인 경우에는 두 슬롯 중에서 한 곳에 새로운 데이터가 삽입되지만 저장 장소를 옮겨야 하는 노드가 비어있는 저장 장소를 찾지 못할 경우에는 계속적으로 새로운 해시 함수를 적용하여 삽입 연산을 재시도해야 하는 문제가 발생한다. 따라서 쿠쿠 해싱 기법은 데이터 검색에는 최악의 상황에도 상수 시간을 보장하지만 데이터 삽입 지연 시간을 예측할 수 없는 문제를 가지고 있다. 정리하면 쿠쿠 해시 기법은 구현의 편리함, 안정적인 성능, 메모리의 효율적 사용 등의 장점이 있으나, 삽입 작업의 오버헤드가 크고 적합한 해시 함수가 알려지지 않은 점, 삽입 과정에서 싸이클이 발생하여 리해싱(Rehashing)이 자주 필요한 점 등의 단점이 있다.

2.3.4 구글 해시

구글 해시는 가장 널리 사용되고 있는 해시 기법 중의 하나이다. 구글 해시는 구글 덴스 해시(Google Dense Hash)와 구글 스파스 해시(Google Sparse Hash)로 나누어져 있는데, 덴스 해시는 Quadratic Probing 기법을 사용하는 일반적인 해시 테이블 구현으로 알려져 있다. 구글 스파스 해시는 덴스 해시를 기반으로 스파스 해시 기법을 구현하였다. 스파스 해시 기법은 해시 테이블을 논리적으로 다수의 그룹으로 분할하며 그룹 별로 메모리의 여러 곳에 분산되어 저장한다.

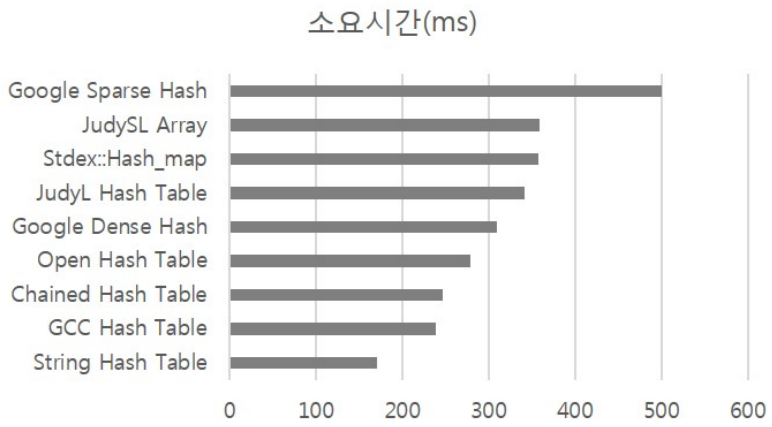

그림 2.9 해시 테이블 벤치마크

각 그룹의 크기는 상수 개의 버킷으로 고정되어 있으므로 데이터의 삽입이 상수 시간에 이루어진다. 또한 논리적인 위치와 스파스 테이블 위치를 사상하는 비트맵에 데이터가 저장되어 있는지 유무를 저장하여 데이터의 검색 역시 상수 시간에 이루어진다[40]. 이 기법은 분할된 각 그룹의 크기를 크게 지정할수록 각 스파스 테이블의 속도는 느려지지만 위치를 사상하는 데 필요한 오버헤드는 줄어드는 특징이 있다.

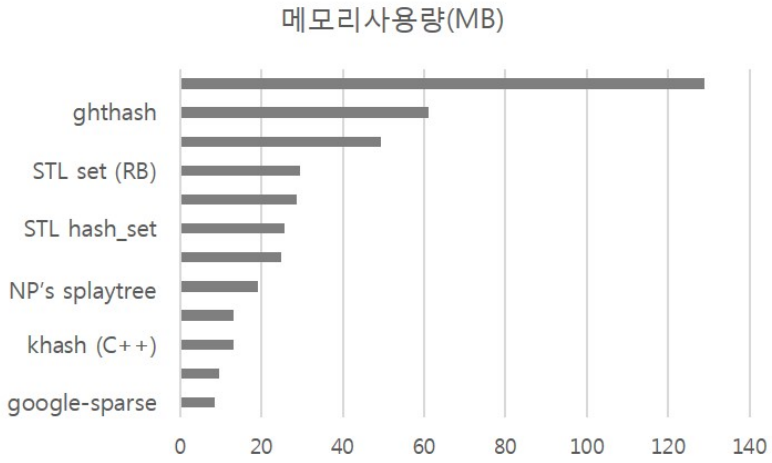

그림 2.10 해시 테이블 벤치마크

다양한 해시 기법의 성능을 평가한 결과에 따르면, 구글 덴스 해시의 성능은 평균적이며, 구글 스파스 해시는 다른 해시 기법에 비해 속도는 느리지만 안정적인 성능을 제공하고 특히 메모리 공간 효율성이 매우 높다고 알려져 있다[41].

제 3 장 시스템 모델

본 장에서는 먼저 제안 기법을 적용하고자 하는 시스템의 구성에 대해 설명한다. 먼저 응용을 포함한 전체 시스템 구성에 대해 설명하고, 제안하는 메모리 데이터 관리 기법의 장점, 구성 및 데이터의 흐름에 대해 설명한다.

3.1 시스템 모델

메모리 데이터 관리는 최근 대규모 데이터를 처리하는 시스템에서 많이 활용되고 있다. 대규모 데이터를 처리하는 시스템의 경우 그림 3.1과 같이 클라이언트 기기, 응용 서버, 저장 계층으로 구성되어 있다. 응용 서버는 응용의 목적에 따라 다른 형태의 로직 (Business Logic)을 제공하게 된다. 응용이 복잡해지면서 최하위 계층인 데이터 관리 시스템의 종류가 다양해지고 있다. 특히 성능의 요구 수준이 높아짐에 따라, 디스크 기반의 데이터 관리 시스템만으로는 한계가 있으며 응용 서버의 부하를 줄이기 위한 방안으로 메모리를 활용한 데이터 관리 시스템이 제안되었다.

효율적인 메모리 데이터 관리 기법을 통하여 데이터 관리 시스템의 소프트웨어 구조를 단순화 하는 것은 다양한 측면에서 비용을 절감할 수 있다. 구조상으로 더 적은 수의 계층과 컴포넌트가 존재하면 시스템 구축비용이 줄어들고 속도는 향상된다. 또한 시스템의 운영이 쉬워지며 시스템 규모를 유연하게 변경할 수 있고 오류 발생 가능성도 줄어든다. 적절한 메모리 데이터 관리 기법을 사용할 경우 응용 프로그램의 크기도 상당히 줄일 수 있다. 하드웨어 측면에서도 고성능의 디스크 기반 시스템과 메모리 기반 데이터 관리 시스템의 초기 및 운용비용을 비교할 경우 상당한 차이를 보인다. 고성능 디스크 기반 시스템 구축 시 다른 디스크에 중복된 데이터를 저장하는 방식으로 데이터 전송률을 끌어올리기 때문에 디스크 공간의 낭비와 일관성 관리에 상당한 노력이 필요하다. 따라서

디스크 자체의 가격은 싸더라도 운용비용을 고려한다면 메모리 기반 데이터 관리 시스템이 더 효율적이다.

이러한 메모리 기반 데이터 관리 기술은 소프트웨어 구조와 기본적인 소프트웨어 개발의 패러다임뿐만 아니라 하드웨어 구성과 같은 시스템 전반에 영향을 끼친다. 여러 질의 결과를 캐싱한 중복된 데이터 뷰는 분석 질의의 응답 속도를 최적화하기 위해서 사용된다. 이러한 중복된 데이터 뷰의 크기를 줄이는 것은 많은 오버헤드를 발생 시킨다. 이러한 오버헤드는 빠른 응답시간을 요구하거나 실시간으로 대용량의 데이터를 처리할 경우 더 커진다. 여러 계층으로 구성된 복잡한 데이터 관리 시스템에 데이터 집약적인 작업을 요청하는 것보다 단순화된 데이터 관리 시스템으로 전환하는 것이 대규모의 트랜잭션 유발과 데이터 관리 시스템의 과부하를 줄이는데 큰 역할을 한다.

본 논문의 시스템 모델은 그림 3.2와 같이 저장 계층에서 디스크 기반 시스템이 아닌 메모리 기반 시스템을 사용하는 경우를 가정한다. 메모리 기반 데이터 관리 시스템은 마스터 서버, 데이터 서버, 백업 서버로 구성된다. 본 논문에서 제안하는

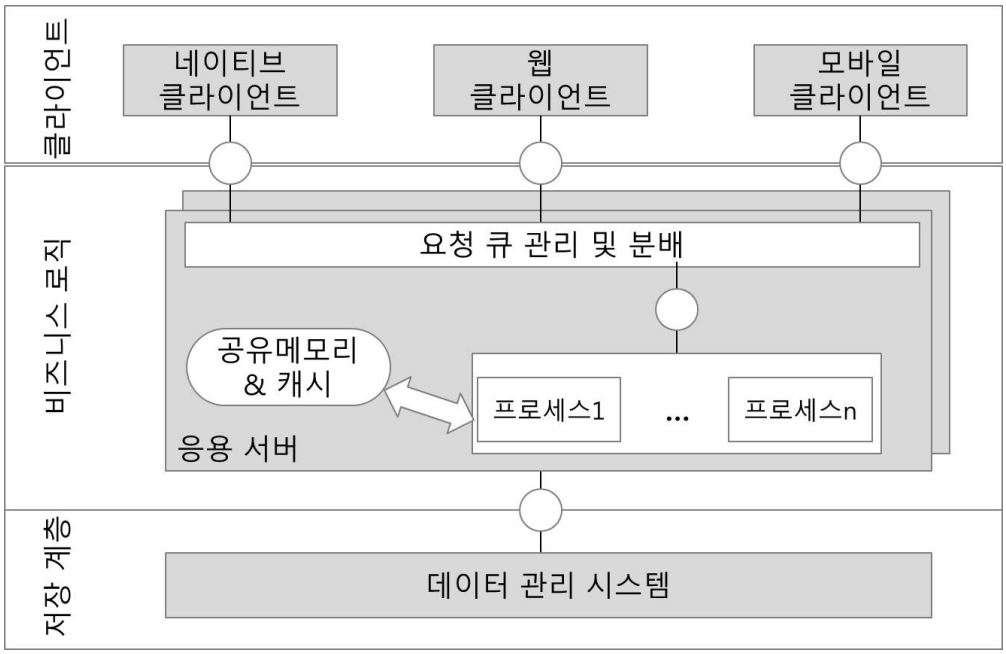

그림 3.1 데이터 관리 시스템의 구조

메모리 데이터 관리 기법은 그림 3.1에서 마스터 서버가 데이터 서버로 데이터를 분배하고, 데이터 서버에서 해시 알고리즘으로 이를 관리하는 방법, 각 데이터 서버가 백업 서버에 사본을 백업하고 복구하는 기법을 다룬다. 응용으로부터 데이터 접근 요청이 들어오면 마스터 서버가 데이터를 저장할 서버를 결정하여 데이터 서버로 보내게 된다. 각 데이터 서버에 저장되는 데이터는 중복되지 않으며, 데이터 서버는 해시테이블을 사용하여 저장된 데이터를 관리하고 데이터 접근 요청에 대해 서비스를 제공한다. 또한 각 데이터 서버는 데이터 서버와 1:N 대응되는 백업 서버에 저장된 데이터를 분산 백업하여 시스템 장애 발생 시 데이터 서버를 복구한다.

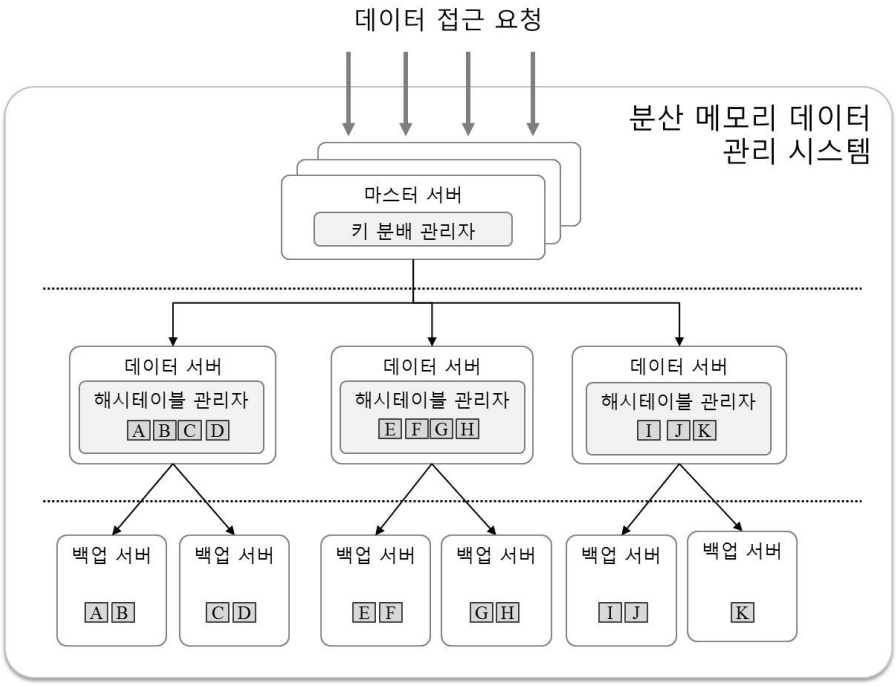

그림 3.2 메모리 기반 데이터 관리 시스템 구조

3.2 메인 메모리 데이터 분배

사용자로부터 데이터 접근 요청이 들어오면 마스터 서버는 기존의 Consistent 해시 기법을 사용하여 데이터를 다수의 데이터 서버에 분산 적재 및 관리한다. 만약 N 개의 캐시 역할을 하는 노드가 있다고 가정하고 이때 부하 분산에 사용하는 일반적인 방법은 오브젝트 o 다음과 같은 해시 함수를 사용하여 $hash(o)$ mod N 번째 노드에 저장하는 기법으로 동적으로 노드가 추가되거나 제거되기 전까지는 정상적으로 동작한다. 그러나 노드가 추가되거나 장애로 제거 되었을 경우, N이 변경되게 되면 모든 오브젝트는 새로운 위치에 모두 재할당을 해야 하는데 이러한 작업을 처리하는 것은 오버헤드가 크다.

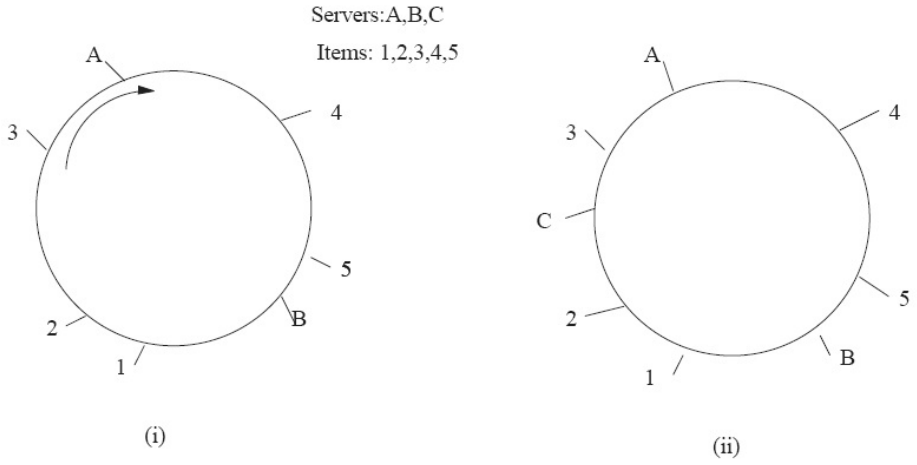

그림 3.3 Consistent 해싱 기법의 동작 원리

Consistent 해시 기법을 사용하게 되면 동적으로 노드가 추가되거나 작동이 중단되어 노드가 제거되어도 모든 오브젝트를 재할당을 하는 것이 아니라, 추가되면 인접한 다른 노드에서 적정한 양의 오브젝트를 전달 받게 되고 마찬가지로 제거된다면 남은 주변의 노드들이 나누는 형태가 되어 일관되게 일부의 오브젝트에 대해서 재할당을 수행하게 된다. 따라서 기존의 노드들에 저장된 대부분의 캐시를 사용할 수 있으며 시스템 변동이 캐시 적중률에 영향을 미치지 않게 된다. Consistent 해시 기법은 오브젝트와 노드가 모두 동일한 해시 함수를 사용해서

해싱 한다. 노드는 구간을 정하고 그 구간에는 적합한 오브젝트의 해시 값을 가지고 있는다. 노드가 제거 되면 인접한 구간의 노드가 제거된 노드의 구간을 위탁 받고 다른 노드들은 제거된 노드로부터 영향을 받지 않는다.

그림 3.3은 링 구조를 활용한 Consistent 해시 기법의 동작 원리를 보여 준다 [32]. 각 아이템 1, 2, 3, 4, 5는 표준 해시 함수를 통해 원 위의 절대 위치에 사상되고 노드 간의 링크를 연결한 원을 따라 시계 방향으로 가장 근접한 노드에 할당된다. 그림 (i)에서 A와 B 노드만 존재한다고 했을 때 노드 A는 아이템 1, 2, 3을 저장하고, 노드 B는 아이템 4, 5를 저장한다. 그림 (ii)에서 노드 C가 시스템에 추가 되었을 때 노드 C는 아이템 1, 2를 새로 할당 받고 다른 노드에 저장된 아이템들은 변경 되지 않는다.

제 4 장 효율적인 해시 기법

3장의 시스템 모델 상에서 각 데이터 서버는 해시 기법을 사용하여 데이터를 저장하고 관리한다. 본 장에서는 데이터 서버에서 데이터를 효율적으로 관리하기 위해 충돌을 최소화하고 충돌로 인한 비용을 줄일 수 있는 해시 기법을 제안한다. 제안하는 해시 기법 모델을 제시한다.

4.1 해시 기법 개요

기존의 해시 함수를 사용하여 데이터의 저장 위치를 지정할 경우 동일한 메모리 위치에 데이터가 저장되면 충돌이 발생한다. 기존 메인 메모리 데이터 관리 기법은 체이닝, 밀어내기 등의 기법을 사용해 충돌을 해소하고 있지만 검색 또는 삽입의 성능이 떨어지는 문제점이 있다. 즉 워크로드의 패턴에 따라 메모리의 데이터 접근 처리 시간이 느려지게 된다. 따라서 본 절에서는 두 개의 메인 해시 테이블과 하나의 보조 해시 테이블을 사용하여 해시 테이블에서 데이터의 충돌을 최소화하고 검색과 삽입 연산을 상수 시간에 처리할 수 있는 해시 기법을 제안한다.

그림 4.1은 제안 해시 기법의 구조를 나타낸다. 제안 해시 기법은 두 개의 메인 해시 테이블과 보조 해시 테이블을 함께 사용한다. 두 개의 메인 해시 테이블은 각 슬롯에 하나의 데이터가 저장되며, 보조 해시 테이블은 각 슬롯에 연결 리스트를 저장하여 다수의 데이터 노드를 저장할 수 있다. 데이터는 각 테이블마다 하나씩, 세 개의 슬롯 중 한 곳에 저장된다. 두 메인 해시 테이블의 크기는 동일하며, 보조 해시 테이블은 메인 해시 테이블 크기의 약수로 유지된다. 따라서 두 개의 해시 함수를 사용하여 세 개의 테이블을 관리할 수 있다. 편의를 위해 이하 본문에서는 각 테이블을 테이블 $T1$, $T2$, $T3$로 지칭한다.

또한 제안 해시 기법에서는 각 테이블에서 슬롯마다 데이터가 저장되어 있

는지 아닌지를 판단하기 위해 블룸 필터를 사용한다. 블룸 필터는 둘 이상의 해시 함수를 사용하여 데이터의 해시 값들을 비트맵에 사상하고 해당 요소가 테이블에 존재하는지 아닌지를 검사한다. 데이터 존재 유무를 간편하게 판별할 수 있다는 장점으로 인해 해시 테이블 등의 검색 성능 개선을 위해 자주 사용된다. 단, 거짓인데 참인 경우(False Negative)는 존재하지 않으나 참인데 거짓인 경우(False Positive)가 존재한다[43][44].

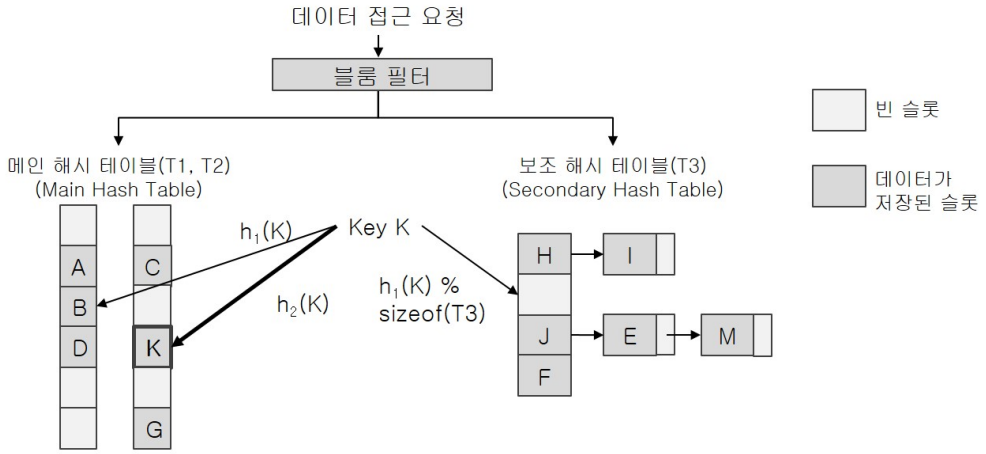

그림 4.1 제안하는 해시 기법

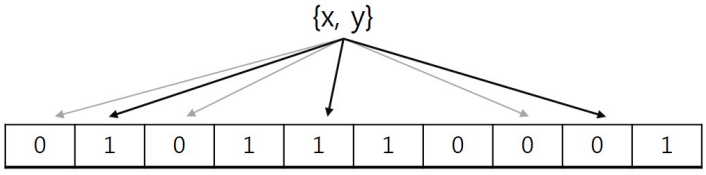

그림 4.2 블룸 필터

제안 해시 기법은 일반적인 해시 테이블과 같이 검색(Lookup), 삽입(Insert), 갱신(Update), 삭제(Delete) 를 기본 연산으로 지원하며, 이에 더해 동적 재배치(Dynamic Relocation) 연산을 수행한다. 각각의 연산에 대한 자세한 설명은 다음 절에서 다룬다.

- Insert(k, v): 키 k와 데이터 v를 보조 해시 테이블에 추가한다.
- Lookup(k): 키 k가 메인/보조 해시 테이블에 있으면 참을, 그렇지 않으면 거짓을 리턴한다.
- Delete(k): 메인/보조 해시 테이블에서 키 k를 찾아 삭제한다.
- Update(k, v): 메인/보조 해시 테이블에서 키 k를 찾아 그 데이터를 v로 갱신한다.

동적 재배치 연산은 보조 해시 테이블의 데이터를 메인 해시 테이블로 이동시키는 작업으로, 시스템의 유휴 시간에 수행된다. 그림 4.3과 같이, 해시 테이블 관리자는 마스터 서버가 요청한 삽입, 삭제, 갱신, 삭제 연산을 각각 하나의 이벤트로 보고 이벤트 큐에 저장한다. 각 연산은 큐에 들어온 순서대로 수행하며, 수행 중 큐가 비는 시점이 있으면 동적 재배치 이벤트가 큐에 삽입되어 수행된다. 동적 재배치는 하나의 데이터 오브젝트를 단위로 수행되며 사용자 요청에 의한 다른 이벤트가 큐에 들어올 때까지 반복한다.

제안 기법의 메인 해시 테이블은 기존의 쿠쿠 해시 기법[47]을 사용하며, 보조 해시 테이블은 체인 해시 기법[34]을 사용한다. 체인 해시 테이블을 쿠쿠 해시 테이블의 지연 쓰기 버퍼와 같이 활용하여, 쿠쿠 해시 기법의 단점인 쓰기 지연 시간과 체인 해시 기법이 읽기 연산이 상대적으로 느린 단점을 상호 보완하는 것이 제안 기법의 특징이다. 제안 기법은 찾는 데이터가 메인 해시 테이블에 존재하는 경우 최대 2번의 참조만으로 성공이 보장되므로 최악의 경우를 보장할 수 없는 체인 해시 기법에 비해 안정적이다. 또한 보조 해시 테이블을 사용하여 데이터를 삽입하므로 기존의 쿠쿠 해시 기법에 비해 응답 시간이 지연되는 현상을

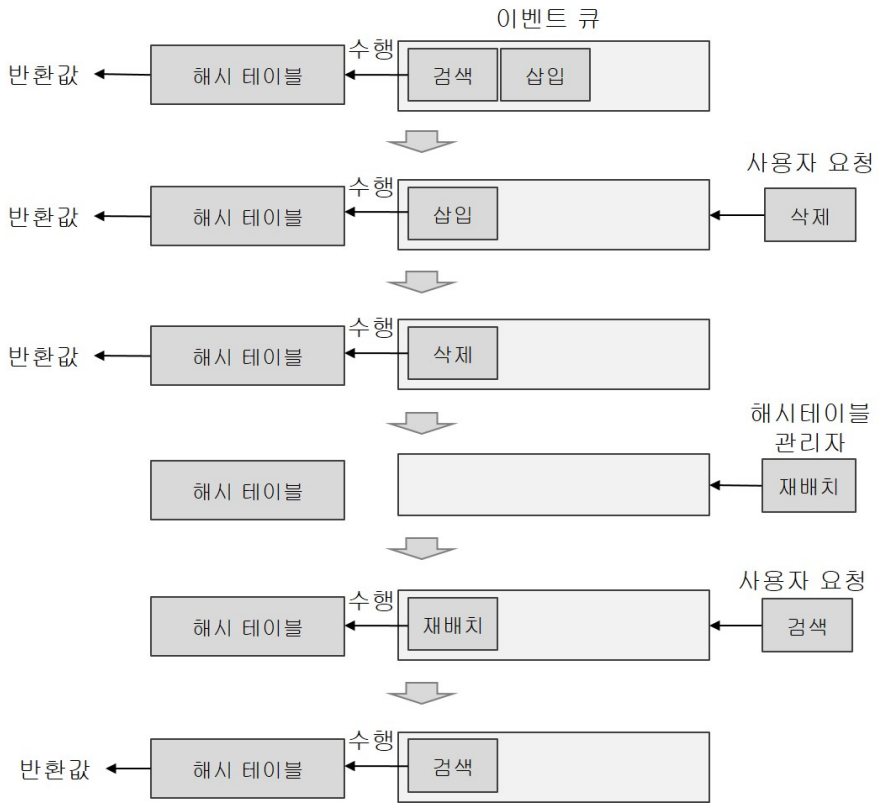

그림 4.3 이벤트 큐 관리

현저히 줄일 수 있다. 따라서 제안 기법은 기존 데이터 관리 기법에 비해 데이터의 충돌을 최소화하여 메모리 데이터를 효율적으로 관리함으로써 시스템의 성능을 높이게 된다.

4.2 해시 기법 동작 과정

본 절에서는 제안하는 해시 기법에서 검색, 삽입, 삭제, 갱신 연산과 동적 재배치 연산의 수행 과정을 설명한다.

4.2.1 삽입

제안하는 해시 기법에서 데이터는 메인 해시 테이블 또는 보조 해시 테이블에 저장된다. 삽입 연산의 처리 과정은 다음과 같다. 제안 해시 기법에서 키 K 가

해시 테이블 내에 없다는 가정 하에 키 K를 삽입한다고 할 때, 키 K는 테이블 $T1$에서 $h_1(K)$, 테이블 $T2$에서 $h_2(K)$, 보조 테이블 $T3$에서 ($h_2(K)$ 를 T1의 크기로 나눈 나머지) 세 슬롯에 저장될 수 있다. 이 때 다음 두 가지 경우가 발생한다.

- 메인 해시 테이블 $T1$, $T2$의 두 슬롯 중 하나 이상이 비어 있는 경우
- 메인 해시 테이블 $T1$, $T2$의 두 슬롯에 모두 다른 데이터가 저장되어 있으며, 보조 해시 테이블 $T3$의 슬롯이 비어 있는 경우
- 메인 해시 테이블 $T1$, $T2$의 두 슬롯에 모두 다른 데이터가 저장되어 있으며, 보조 해시 테이블 $T3$의 슬롯이 비어 있지 않은 경우

키 K를 삽입하기 위해서는 먼저 첫 번째 경우를 확인한다. 메인 해시 테이블 $T1$, $T2$에서 $h_1(K)$ 또는 $h_2(K)$를 확인한다. 예를 들어 그림 4.4의 경우 테이블 $T1$의 슬롯은 데이터가 들어 있고 테이블 $T2$의 슬롯이 비어 있으므로 테이블 $T2$에 키 K를 삽입한다.

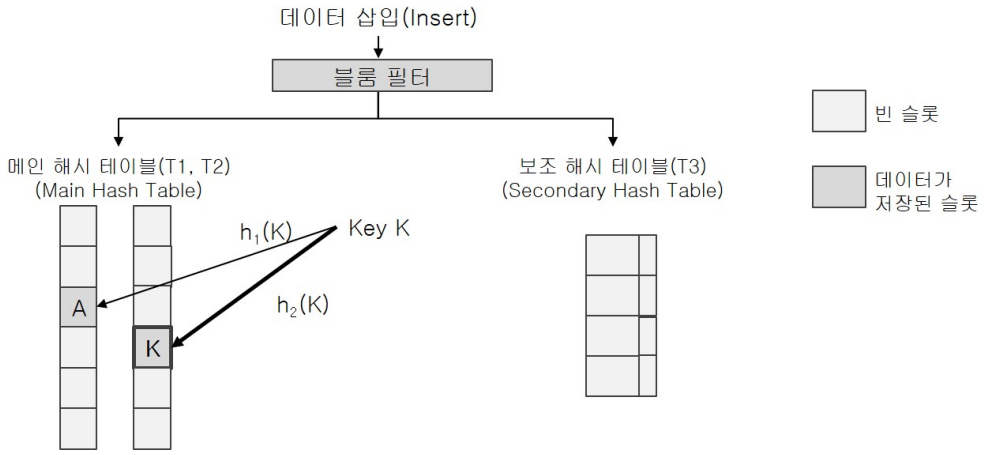

그림 4.4 키 삽입: T1, T2에 빈 슬롯이 존재

만약 메인 해시 테이블의 두 슬롯에 모두 다른 데이터가 저장되어 있다면 보조 해시 테이블 $T3$의 슬롯을 찾는다. $T3$의 슬롯이 비어 있다면 키 K를 해당

위치에 삽입하고, 비어 있지 않다면 키 K를 이 슬롯의 리스트 헤드로 만들어 삽입한다. 예를 들어 그림 4.5는 테이블 $T1$의 슬롯과 테이블 $T2$의 슬롯에 모두 데이터가 들어 있는 상태이므로 키 K를 보조 해시 테이블의 슬롯에 삽입하며, 그림 4.6은 세 테이블의 슬롯에 모두 데이터가 들어 있는 상태이므로 키 K를 보조 해시 테이블의 슬롯의 리스트 헤드로 삽입한다.

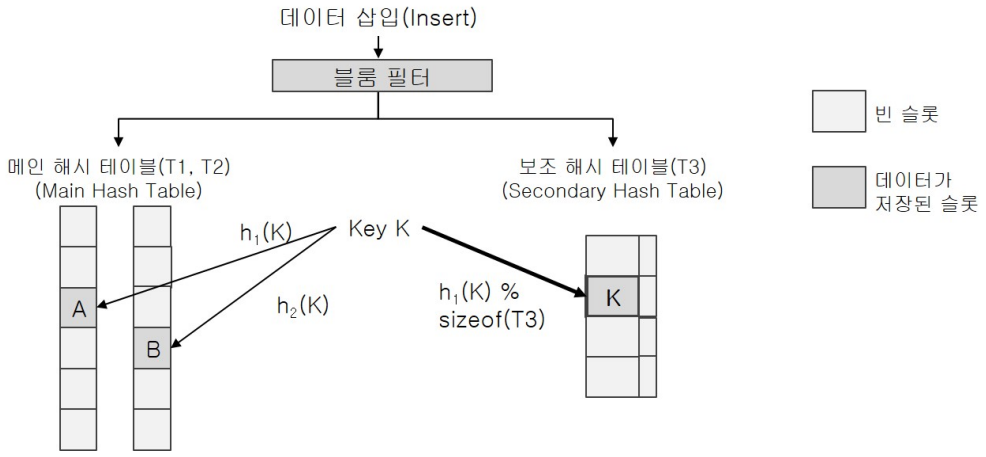

그림 4.5 키 삽입: T1, T2는 차 있고 T3의 슬롯은 비어 있음

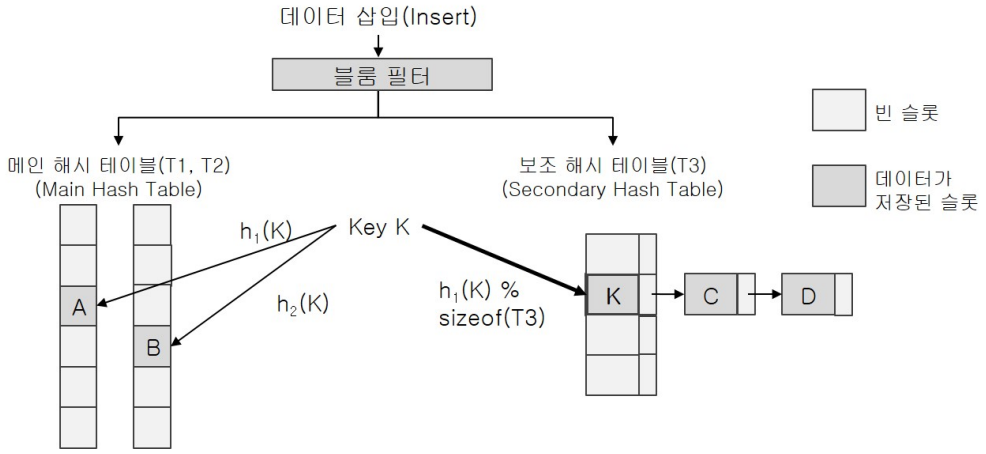

그림 4.6 키 삽입: T1, T2, T3의 슬롯에 모두 데이터가 있음

위 처리 과정에서 해시 테이블의 유무를 판별할 때, 필터의 반환값이 거짓인 경우 바로 삽입할 수 있으므로 메인 해시 테이블에 접근해야 한다. 반환값이 참인 경우에도 False Positive가 존재할 수 있으므로 메인 해시 테이블의 슬롯을 직접 접근해야 한다. 따라서 삽입 시에는 반환값에 관계없이 메인 해시 테이블에 접근해야 하므로, 삽입 전 필터 값을 확인할 필요는 없고 삽입 후에 필터의 비트를 설정하는 작업만을 수행한다.

알고리즘 4.1은 삽입 연산의 수행 과정을 나타낸 의사코드이다.

Algorithm 4.1 키 삽입 과정 의사 코드

```
function INSERT(K)
    if T1[h1(K)] is empty then
        T1[h1(K)] ← K
    else
        if T2[h2(K)] is empty then
            T2[h2(K)] ← K
        else
            T3[h1(K) mod sizeof(T3)] ← K
        end if
    end if
end function
```

4.2.2 검색, 갱신 및 삭제

제안하는 해시 기법의 검색 연산은 메인 해시 테이블 $T1, T2$를 먼저 검색한 후 보조 해시 테이블을 검색한다. 키 K 검색을 위해 먼저 $h_1(K)$와 $h_2(K)$를 사용하여 메인 해시 테이블 $T1, T2$의 슬롯에 접근한다. 이 때 메인 해시 테이블의 상태는 다음 중 하나이다.

- 메인 해시 테이블 $T1, T2$의 두 슬롯이 모두 비어 있는 경우
- 메인 해시 테이블 $T1, T2$의 두 슬롯 중 하나에만 데이터가 존재하는 경우
- 메인 해시 테이블 $T1, T2$의 두 슬롯 모두에 데이터가 존재하는 경우

먼저 그림 4.7과 같이 두 슬롯이 모두 비어 있는 경우, 필터를 통해 이를 확인할 수 있다. 따라서 보조 해시 테이블로 바로 접근하여 리스트를 순차 검색한다.

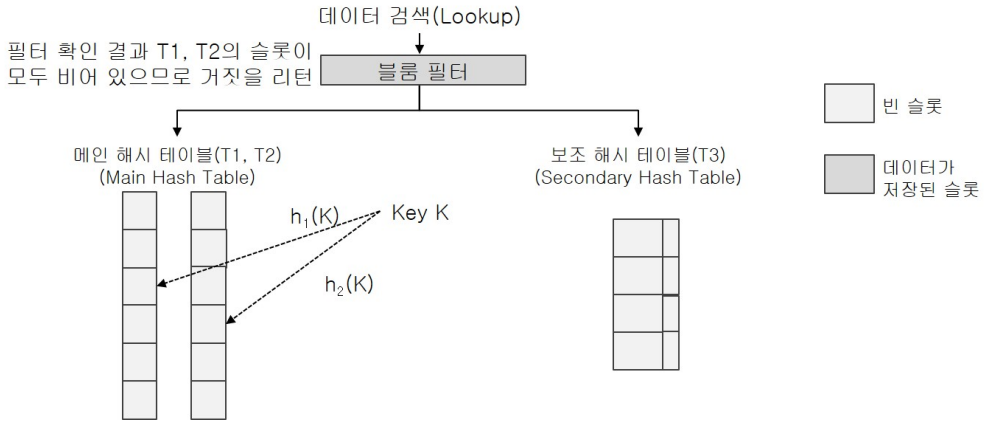

그림 4.7 키 검색: T1, T2의 두 슬롯 모두 비어있음

두 번째로 메인 해시 테이블 $T1$ $T2$의 두 슬롯 중 하나에만 데이터가 존재하는 경우, 데이터가 저장된 슬롯을 확인한다. 그림 4.8과 같이 해당 슬롯에 저장된 키가 K 이면 참을 반환하고, 그림 4.9와 같이 다른 키카 저장되어 있으면 보조 해시 테이블의 리스트를 순차 검색한다.

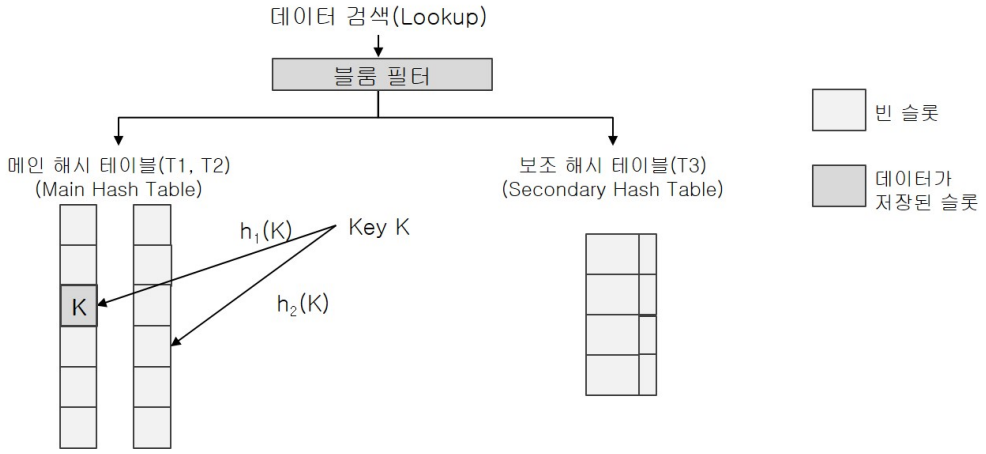

그림 4.8 키 검색: T1, T2의 슬롯 하나에만 데이터가 존재 (1)

세 번째로 두 슬롯 모두에 데이터가 존재하는 경우, 그림 4.10과 같이 메인 해시 테이블의 두 슬롯 중 하나에 키 K 가 있으면 참을 반환한다. 둘 다 K 가

아니라면 키 K가 보조 해시 테이블 T3에 있는지를 확인해야 한다.

보조 해시 테이블 T3의 슬롯을 확인하는 과정은 다음과 같다. 먼저 필터의 값을 확인한다. False Negative는 없으므로, 그림 4.11과 같이 필터 값이 거짓이면 보조 해시 테이블에 접근하지 않고 검색 연산의 결과로 거짓을 반환한다. 필터 값이 참이면 보조 해시 테이블 슬롯의 리스트를 순차 검색하여 키 K가 있는지 확인한다.

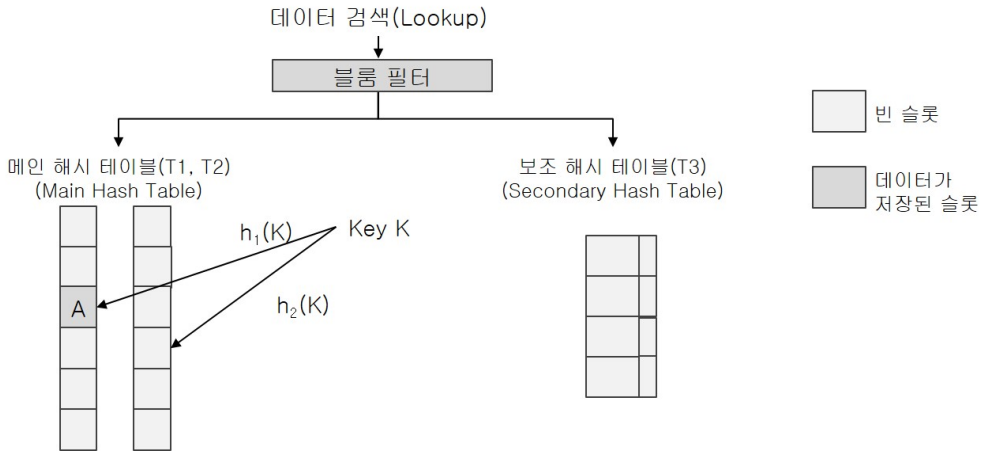

그림 4.9 키 검색: T1, T2의 슬롯 하나에만 데이터가 존재(2)

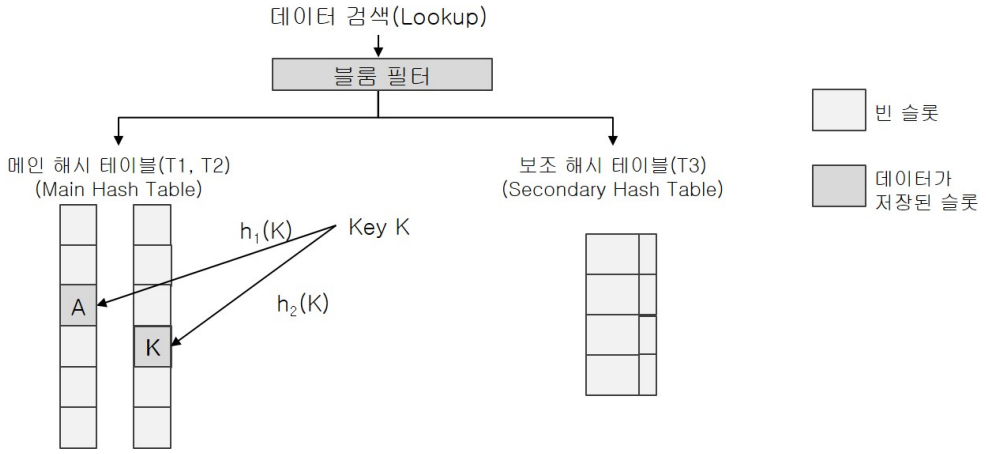

그림 4.10 키 검색: 두 슬롯이 모두 차 있고 키 K를 찾음

만약 키 K를 찾으면 키 K를 리스트의 헤드로 이동시킨 후 참을 리턴한다 (그림 4.12, 4.13). 이 기법은 참조 지역성[42]을 활용한 것으로서, 자주 참조될 것으로 예상되는 데이터에 대해 두 가지 효과를 만들어 접근 지연 시간을 단축시킬 수 있다. 먼저 해당 키를 리스트 헤드로 이동시키므로 다음에 참조될 때 리스트의 순차 검색 비용이 감소한다. 또한 같은 리스트 내에 있는 다른 데이터에 비해 우선적으로 메인 해시 테이블로 재배치되게 된다. 알고리즘 4.2는 검색 연산의 수행 과정을 나타낸 의사코드이다.

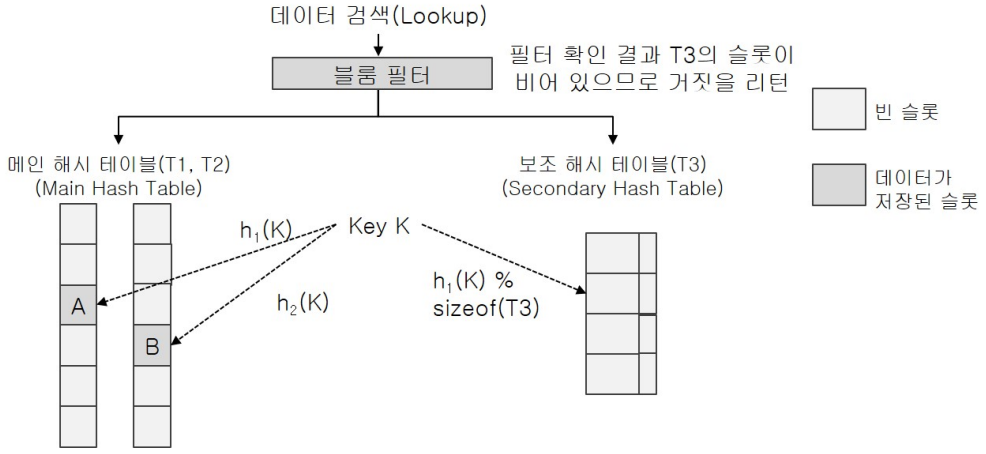

그림 4.11 키 검색: T3의 슬롯이 비어 있음

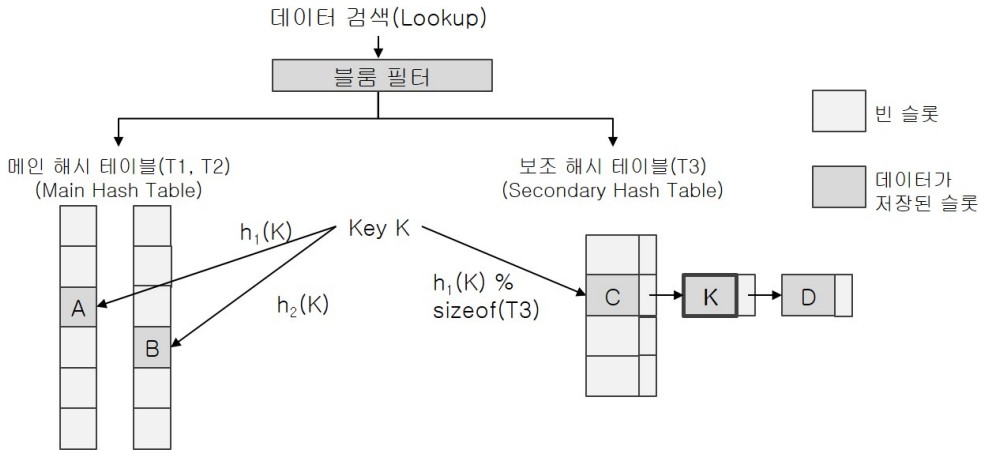

그림 4.12 키 검색: T3 슬롯의 리스트 검색

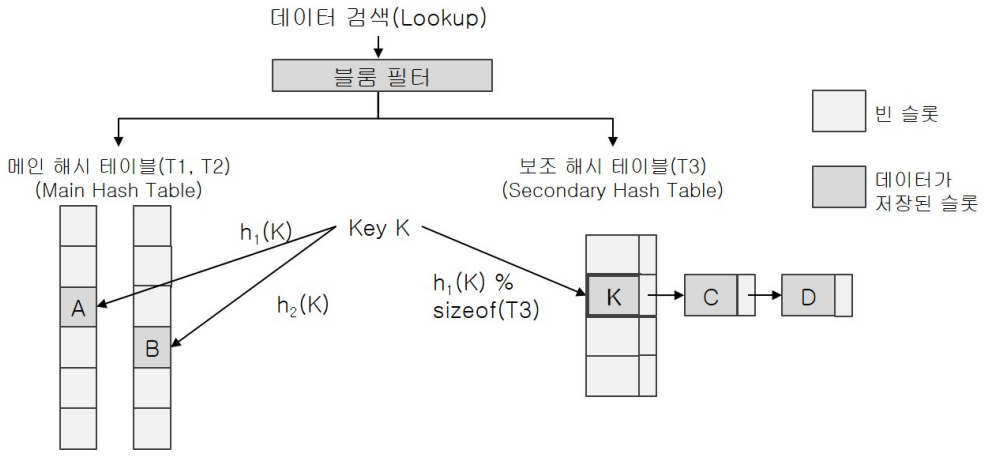

그림 4.13 키 검색: T3 슬롯의 리스트 헤드로 이동

Algorithm 4.2 키 검색 과정 의사 코드

```
function LOOKUP(K)
    if T1[h1(K)]isempty AND T2[h2(K)] is empty then
        return ListSearch(T3[h1(K) mod sizeof(T3)])
    else
        if T1[h1(K)] is occupied AND T2[h2(K)] is occupied then
            if T1[h1(K)]=K OR T2[h2(K)]=K then
                return true
            else
                return ListSearch(T3[h1(K) mod sizeof(T3)])
            end if
        else                                    ▷ only 1 slot is occupied
            if T1[h1(K)]=K OR T2[h2(K)]=K then
                return true
            else
                return ListSearch(T3[h1(K) mod sizeof(T3)])
            end if
        end if
    end if
end function
```

갱신(Update)과 삭제(Delete) 연산은 해당 데이터의 키가 해시 테이블에 존재하는지 먼저 검색한 후, 존재하면 데이터 오브젝트의 내용을 갱신 또는 삭제 데이터가 해시 테이블에 존재하지 않는다면 거짓을 리턴한다. 따라서 검색(Lookup) 연산과 큰 차이가 없으므로 설명을 생략한다.

4.2.3 동적 재배치

제안하는 해시 기법에서 보조 해시 테이블에 삽입된 데이터는 시스템이 유휴 상태일 때 메인 해시 테이블로 동적으로 재배치된다. 데이터가 보조 해시 테이블에 있을 때보다 메인 해시 테이블에 있을 때 검색의 효율이 높아지므로 이와 같은 동적 재배치를 통해 시스템의 응답성을 높일 수 있다.

4.1 절에서 언급한 바와 같이, 동적 재배치 연산은 보조 해시 테이블에서 임의의 키를 메인 해시 테이블로 이동시킨다. 재배치할 키를 선택한 후 메인 해시 테이블의 상태를 두 가지로 나눠 생각해볼 수 있다.

- 메인 해시 테이블 $T1$ 또는 $T2$에 빈 슬롯이 있는 경우
- 메인 해시 테이블 $T1, T2$의 두 슬롯 모두에 데이터가 존재하는 경우

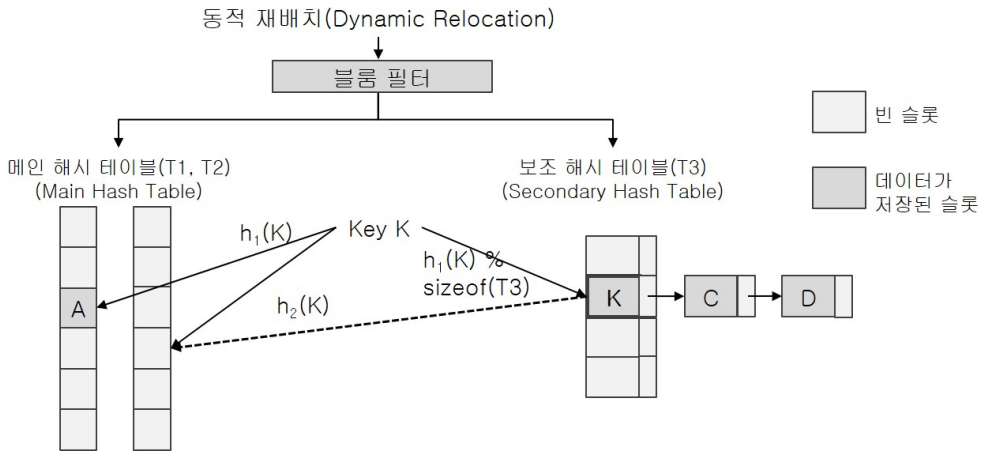

그림 4.14 동적 재배치: 빈 슬롯이 존재

동적 재배치 연산을 제외하면, 메인 해시 테이블에서 데이터의 존재 유무는

삽입, 삭제 연산에 의해 변경된다. 삽입 연산의 경우 메인 해시 테이블의 빈 슬롯에 우선적으로 삽입되며 삭제 연산은 메인 해시 테이블 또는 보조 해시 테이블에 이동시킬 데이터가 존재한다면 삭제한다. 따라서 첫 번째 경우, 그림 4.14와 같이 이전에 삭제 연산에 의해 메인 해시 테이블에 빈 슬롯이 만들어진 상태이므로 되면 보조 해시 테이블에 존재하는 키를 메인 해시 테이블의 빈 슬롯으로 이동시킨다.

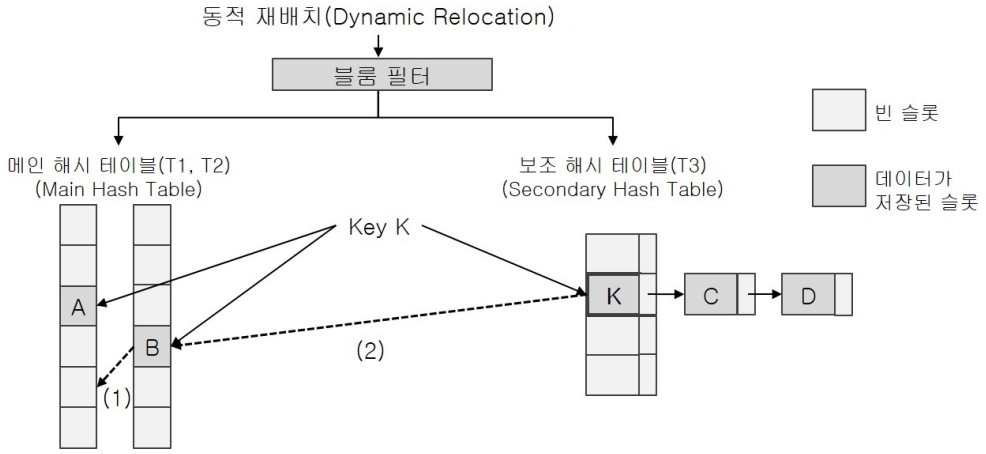

그림 4.15 동적 재배치: 밀어내기(1)

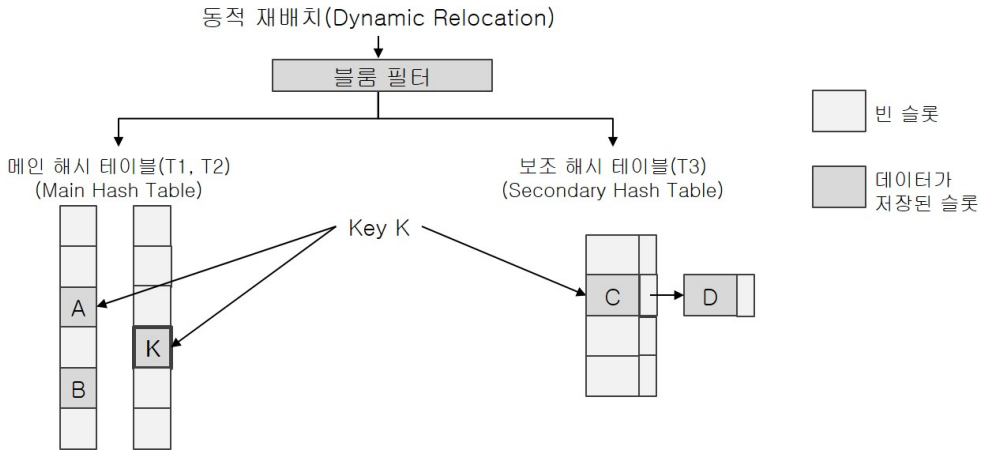

그림 4.16 동적 재배치: 밀어내기(2)

두 번째 경우 먼저 밀어내기 기법을 사용하여 메인 해시 테이블에 빈 슬롯을

만들고 해당 키를 이동시킨다. 예를 들어, 그림 4.15에서 키 K를 재배치하려고 한다. 메인 해시 테이블 $T1, T2$의 슬롯 $h_1(H)$와 $h_2(H)$에 모두 다른 데이터(A, B)가 저장되어 있으므로 키 A와 키 B 둘 중 하나를 다른 테이블로 밀어내야 한다. 이 경우 $T1$에 있는 키 B를 $T2$의 $h_2(B)$로 밀어낸 후 키 K를 리스트 헤드에서 제거하고 $T2$의 빈 슬롯에 재배치한다. 재배치 후의 상태는 그림 4.16과 같다.

이와 같은 밀어내기는 재귀적으로 일어날 수 있다. 예를 들어 그림 4.17과 같은 상태에서는 $T1$의 $h_1(B)$에 키 E가 저장되어 있으므로, 먼저 키 E를 T2의 $h_2(E)$로 밀어내고, 키 B를 $T1$으로 밀어낸 후 마지막으로 키 B가 있던 자리에 키 K를 배치한다. 재배치 후의 상태는 그림 4.18과 같다. 밀어내기 경로의 최대 길이는 임의의 상수에 의해 결정되며, 상수보다 큰 경로가 나타나게 되면 해시 테이블 크기를 증가시키고 데이터 전체를 재배치하는 리해싱(Rehashing) 작업을 진행하게 된다.

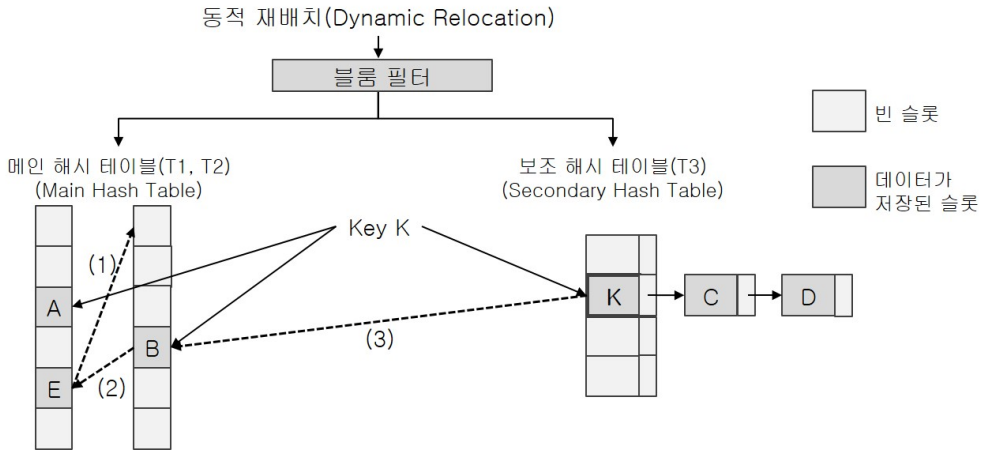

그림 4.17 동적 재배치: 밀어내기(3)

4.2.4 동적 재배치의 문제점

동적 재배치 과정은 리스트 내 데이터 노드 하나를 단위로 이루어진다. 향후에는 동적 재배치를 노드 단위가 아닌 슬롯(리스트) 단위로 처리하면 동적

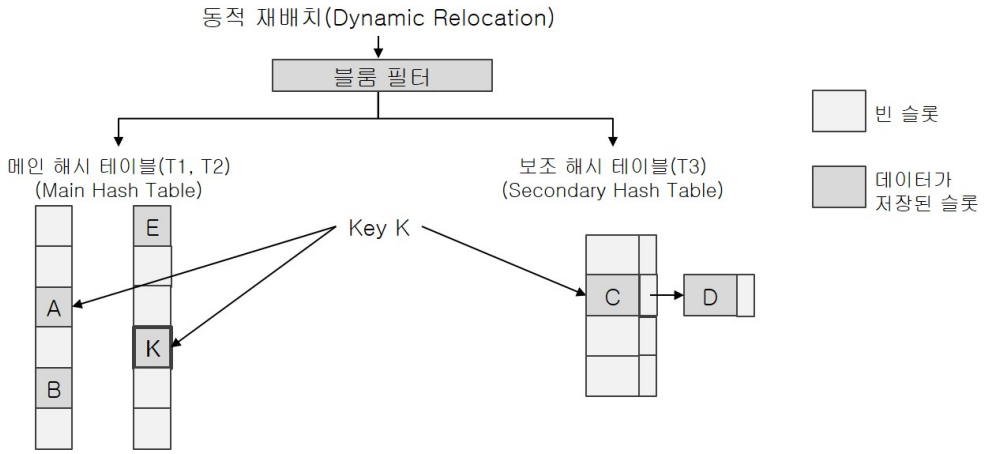

그림 4.18 동적 재배치: 밀어내기(4)

재배치 과정을 효율적으로 구현할 수 있다. 현재 테이블 $T3$ 접근을 위해 메인 해시 테이블 $T1$의 $h_1()$을 사용하고 있는데, 이 방법은 보조 해시 테이블 접근을 빠르게 할 수 있는 반면 같은 리스트 내의 데이터들이 메인 해시 테이블 $T1$에서 충돌을 발생시키기 쉽다는 문제가 있다. 메인 해시 테이블과 보조 해시 테이블의 크기가 같다면 이 충돌 확률은 1이고, 메인 해시 테이블이 보조 해시 테이블의 N배라면 N분의 1 확률로 충돌이 발생한다. 이로 인해 밀어내기 작업 경로가 평균적으로 길어지고 재배치 비용이 커지는 단점이 생긴다. 동적 재배치 과정을 최적화하기 위해, 같은 리스트 내에 있는 다수 노드의 밀어내기 과정들을 한꺼번에 처리하여 가장 효율적인 다중 경로 계산 알고리즘의 연구가 필요하다.

또한 삭제 후 재배치 과정에 대한 연구 또한 필요하다. 메인 해시 테이블의 데이터를 삭제한 후, 보조 해시 테이블 $T3$에서 메인 해시 테이블의 빈 슬롯을 채울 수 있는 데이터를 즉시 이동시킬 수 있다. 예를 들어 그림 4.19에서 키 K를 삭제한 후 $T3$의 슬롯에서 즉시 키 A를 가져와 빈 슬롯을 채우게 한다. 그러면 메인 해시 테이블의 슬롯이 하나 이상 비어 있는 경우 보조 해시 테이블에 접근할 필요가 없어지므로 해시 테이블의 접근 지연 시간이 상당히 단축될 수 있다.

하지만 이 방법은 논리적인 오류가 존재한다. 그림 4.20과 같은 경우, 테이블 $T2$에서 키 K를 삭제하면 $T3$에 있는 키 K의 슬롯은 비어 있으므로 키 K가

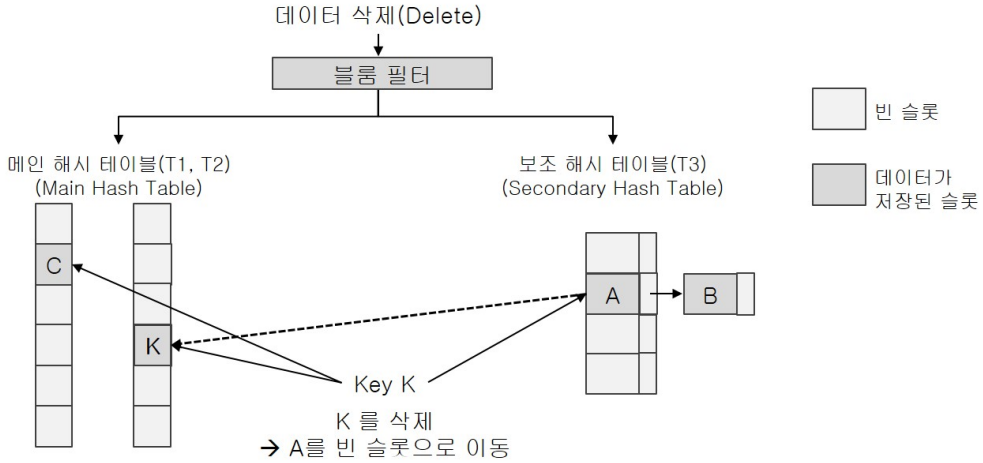

그림 4.19 삭제 후 재배치

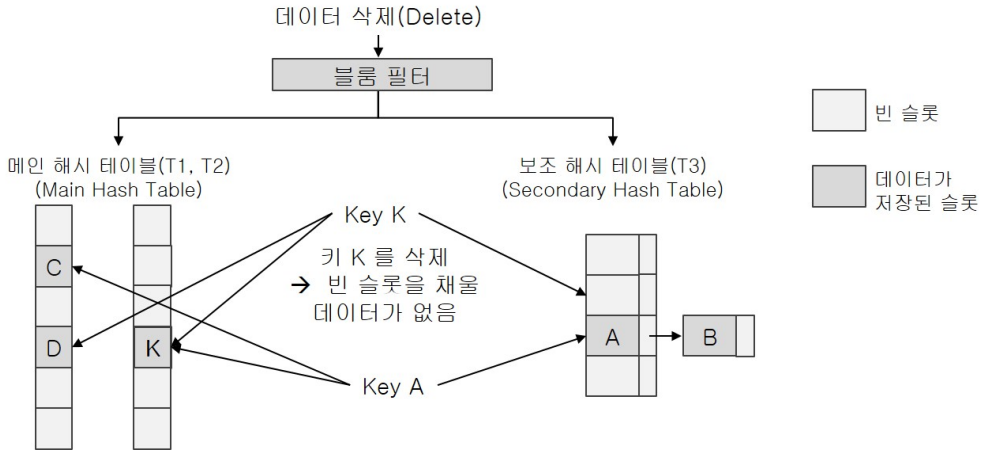

그림 4.20 삭제 후 재배치: 오류(1)

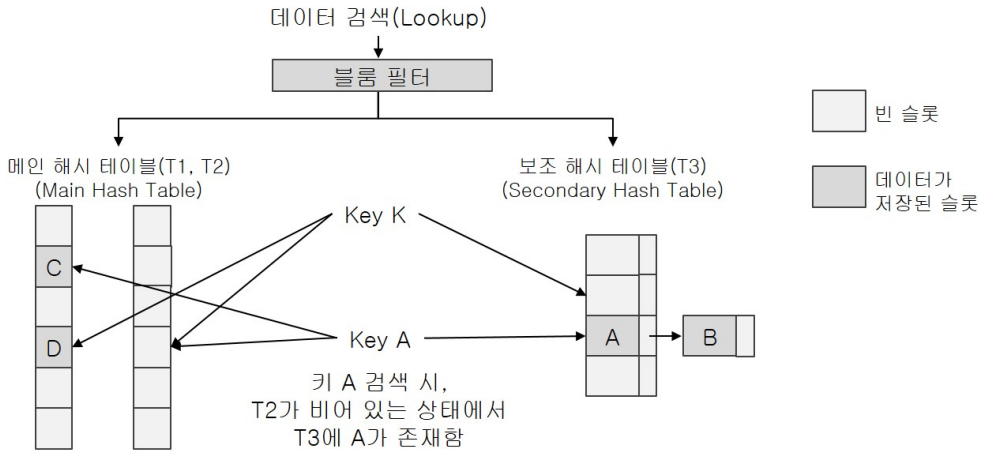

그림 4.21 삭제 후 재배치: 오류(2)

저장되어 있던 $T2$의 슬롯은 빈 상태로 남겨진다. 하지만 그림 4.21과 같이, 키 K 삭제 후 키 A를 검색하게 되면 메인 해시 테이블에 키 A의 빈 슬롯이 존재하는 상태에서 키 A가 보조 해시 테이블에 저장되어 있는 잘못된 상태가 발생할 수 있다. 따라서 이와 같은 문제점을 해결하고 삭제 후 재배치 과정을 정확히 수행하기 위해서는 향후 해시 함수의 재설계가 필요하다.

제 5 장 시간 복잡도 기반 성능 분석

본 장에서는 제안하는 해시 테이블 모델을 제시하고 삽입, 검색 및 삭제 알고리즘과 동적 재배치 과정의 시간복잡도 분석을 통해 제안 기법의 데이터 처리 과정이 상수 시간에 수행됨을 확인한다.

5.1 해시 테이블 모델

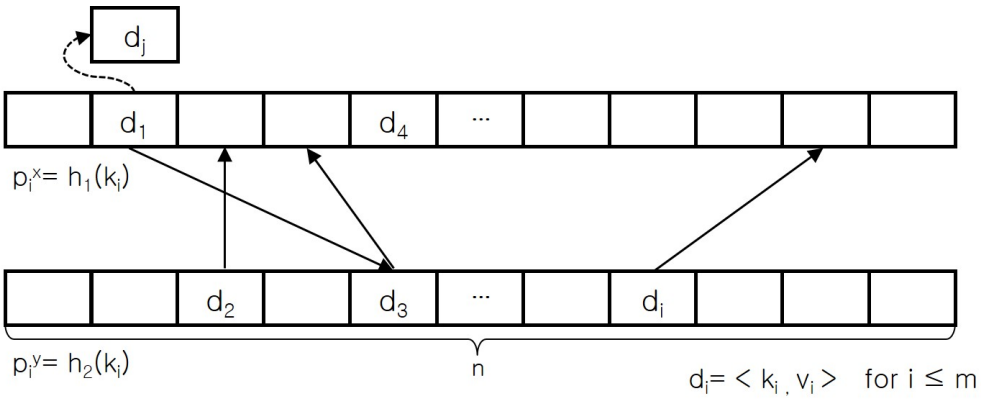

그림 5.1 제안 기법의 해시 테이블 모델

본 논문의 제안 기법은 두 개의 메인 해시 테이블과 하나의 보조 해시 테이블을 유지하여 데이터 노드들을 관리한다. 데이터 노드 d_i는 식별 가능한 k_i와 데이터 v_i쌍으로 구성된다 (단, $i \leq m$). 두개의 메인 해시 테이블에는 같은 데이터 노드의 크기를 가진 슬롯이 n개씩 존재한다. 두개의 해시 함수 h_1, h_2: $\{0,1\}^*$ → $\{1,..., n\}$는 데이터 노드의 키값 k_i을 입력받아 두 개의 메인 해시 테이블 내 위치 $p_i{}^x$와 $p_i{}^y$ (단, $p_i{}^y$, $p_i{}^y \leq n$)를 결정하는데 사용된다. 이 해시함수를 통해 계산된 위치 $p_i{}^x$와 $p_i{}^y$에 해당 데이터 노드가 위치한다. 메인 해시 테이블 각 n개의 슬롯에 대해 하나의 데이터만 위치할 수 있다. 반면 보조 해시 테이블에는 복수개의 데이터가 연결 리스트 방식으로 연결될 수 있다.

표 5.1 시스템 모델을 위한 기호 및 함수

기호 및 함수	설명
n	해시 테이블 내 슬롯의 개수
m	해시 테이블 내 위치한 데이터 노드의 개수
$h_1(x)$	첫 번째 메인 해시 테이블을 위한 해시 함수 $h_1(x) \in \{1, ..., n\}$
$h_2(x)$	두 번째 메인 해시 테이블을 위한 해시 함수 $h_2(x) \in \{1, ..., n\}$
α	해시 테이블의 부하 인자 $\alpha \leq 1$ n 개의 슬롯을 가진 해시테이블에 m 개의 아이템이 있을 때 $\frac{m}{n}$ 로 계산됨
k_i	i 번째 데이터 노드에 대한 식별 가능한 키값 $\{0,1\}^*$
v_i	i 번째 데이터 노드의 값 $\{0,1\}^*$
d_i	i 번째 데이터 노드에 대한 k_i 와 v_i 의 벡터 $d_i = < k_i, v_i >\}$
k	키값들의 집합 $k = \{k_1, k_2, k_3, , \cdots k_m\}$
v	데이터들의 집합 $k = \{v_1, v_2, v_3, \cdots v_m\}$
d	데이터 노드들의 벡터 집합 $d = \{< k_1, v_1 >, < k_2, v_2 >, < k_3, v_3 >, \cdots < k_m, v_m >\}$
$\|d\|$	해시 테이블에 저장된 데이터들의 개수 (즉$\|d\| = m$)

해시 함수 h_1 와 h_2 는 상수 시간에 함수값을 계산한다. 또한 각 데이터 노드들은 다른 해시 값들에 상관없이 슬롯에 갈 확률은 균등하게 모두 같다. 따라서, 임의의 데이터 노드에 대한 해시 테이블에 들어갈 확률은 $1/n$이다.

시스템 모델을 위한 기호 및 함수는 표 5.1과 같다.

5.2 시간 복잡도 기반 성능 분석

본 절에서는 앞의 절에서 설명된 제안 해시 테이블을 사용하는 제안 기법의 삽입, 검색 및 삭제 연산의 시간 복잡도를 분석한다. 제안 기법은 연결 리스트를 저장하는 보조 해시 테이블을 사용하여 해시 알고리즘의 삽입과 검색 속도를 높이고, 시스템의 유휴 시간에 동적 재배치 과정을 수행한다. 단, 검색, 삭제 및 갱신 연산과 달리 동적 재배치 과정은 쿠쿠 해시의 삽입 알고리즘에 기반하므로, 쿠쿠 해시의 삽입 알고리즘의 시간 복잡도 분석과 유사한 흐름을 보이게 된다[54].

데이터 노드 삽입 요청 시 메인 해시 테이블 내 저장 위치를 결정하기 위해

두 번의 해시 값 $h_1(x)$와 $h_2(x)$을 계산한다. 그 메인 해시 테이블의 위치에 비어 있는 슬롯이 있으면 데이터 노드를 삽입한다. 비어 있는 슬롯이 존재하지 않으면 보조 해시 테이블 슬롯의 리스트의 헤드에 삽입한다. 이는 해시 알고리즘에서 빈 슬롯을 찾기 위해 반복적인 작업을 피하기 위해 추가적으로 보조 해시 테이블을 관리하는 이유이다. 따라서 제안 기법의 삽입 연산은 쿠쿠 해시 알고리즘과 다르게 최악의 경우에도 세 번의 해시 테이블 내 데이터 노드 탐색만으로 수행되므로 O(1)이다.

하지만 이러한 보조 해시 테이블에 유지된 데이터 노드들은 검색 연산 수행 시 검색 속도를 감소시킬 수 있다. 메인 해시 테이블에 유지된 데이터들은 두 번의 해시값 계산으로 빠른 상수 시간의 검색 속도를 보이는 반면 연결 리스트에 유지된 데이터에 대한 검색 속도는 최악 상황에서 O(n)의 시간 복잡도를 가지므로 선형적인 성능 감소를 보일 수 있다.

이를 위해 메인 해시 테이블에 상수 시간의 검색 속도를 유지할 수 있도록 보조 해시 테이블에 존재하는 데이터들을 동적으로 재배치하는 작업이 필요하다. 연결 리스트에 위치한 데이터 노드들은 $h_1(x)$와 $h_2(x)$ 위치에 해당하는 슬롯에 다른 데이터에 의해 채워져 있으므로 두 데이터 중 하나를 선택하여 유효한 다른 위치로 이동시킨 후 저장하며, 이 과정은 빈 슬롯을 찾을 때까지 반복 수행한다.

반복적인 동적 재배치 과정의 시간 복잡도 분석을 위해 그림 5.1과 같이 먼저 두 개의 메인 해시 테이블 내 슬롯을 정점으로 하고 삽입된 데이터 노드d_i 들의 두개의 해시 함수값을 간선으로 하는 이분 그래프(bipartite graph)를 고려한다. 즉 이분 그래프 G는 $V = \{1, ..., n\}$이고 E=$\{e_x | e_x = (h_1(x),\ h_2(x),$ 단 $x \in k)\}$로 구성된다. 데이터 노드의 위치가 해시 함수에 의해 1,..., n 범위에서 할당된 랜덤 변수인 사실을 고려할 때 그래프 G는 확률적으로 구성되며 여러 개의 연결 요소(connected component)로 구성된다. 따라서 연결 리스트 상에 있는 d_j 데이터 노드에 대한 반복적인 동적 재배치 알고리즘의 수행 시간은 이분 그래프 G 상에서 그 간선 ($e_j = (h_1(k_j),\ h_2(k_j))$ 상의 어떤 한 점 부터 도달 가능한 경로의 길이 $L(e_j)$로서 계산할 수 있다. $h_1(k_j)$ 또는 $h_2(k_j)$ 부터 계산될 수 있으므로

m 개의 데이터가 삽입된 메인 해시 테이블에서 연결 리스트에 있는 e_j 를 동적 재배치하기 위한 반복적인 시간은 $L(e_j)$ 보다 작다. 따라서, x_j 데이터 노드의 동적 재배치 과정의 시간 복잡도 $\tau_{Insert}(x)$ 가 $O(1)$ 를 보이기 위해서는 수식 (5.1)을 보이는 것으로 충분하다.

$$E[L(x_j)] \leq O(1) \tag{5.1}$$

정리 1 *임의의 상수 $c \geq 1$ 에 대해, $m \leq \frac{n}{2c}$ 를 만족할 때, 제안 기법의 동적 재배치 과정은 O(1)의 시간 복잡도를 가진다.*

증명 데이터 노드 x_j 삽입 연산의 동적 재배치 과정의 시간 복잡도는 이분 그래프에서 연결 요소의 평균적인 길이로 계산할 수 있다. 연결 요소의 평균적인 길이는 데이터 노드 집합 d의 원소들이 연결 요소에 포함될 평균적인 확률의 합과 같다. 각 데이터 노드들이 연결 요소에 포함될 확률은 같으므로 수식 (5.2)와 같이 정리될 수 있다.

$$\begin{aligned} E[L(x_j)] &= \sum_{i}^{m} Pr[e_i \in L(x_j)] \\ &= m \cdot Pr[e_i \in L(x_j)] \end{aligned} \tag{5.2}$$

수식 (5.2)에서 평균적인 동적 재배치 과정의 시간 복잡도를 위해서는 $Pr[e_i \in L(x_j)]$ 을 계산해야 한다. 어떤 해시 테이블의 위치 u와 v에 대해 u와 v 사이에 경로가 생기고 그 연결 요소의 경로 길이가 $\ell \geq 1$ 인 확률은 최대 $\frac{c^{-\ell}}{n}$ 이다. 이 연결 요소의 경로 길이가 $\ell \geq 1$ 인 확률은 다음과 같이 수학적 귀납법으로 증명한다.

초기식 $\ell = 1$ 일 때, 다음과 같이 확률은 $\frac{c^{-\ell}}{n}$ 보다 작음이 만족한다.

$$
\begin{aligned}
Pr[\exists u \rightarrow v\text{: 경로 길이 1}] &= Pr[\{u,v\} \in E] \\
&= Pr[h_1(x) = u \cap h_2(y) = v] \\
&= \frac{2}{n^2}
\end{aligned}
\tag{5.3}
$$

해시 함수가 $\{1,...,n\}$ 범위에서 균등한 분포를 가진다는 가정에 의해 $\exists x Pr[h_1(x) = u] = \frac{1}{n}$ 이다. 또한 v, u 가 바뀔 수 있으므로 확률은 $\frac{2}{n^2}$ 이다. 임의의 두 점 u, v 에 대해 평균적으로 경로가 1인 연결 요소가 발생할 확률은 m 개에 대해서 모두 동일하고 $2 \cdot m \leq \frac{n}{c}$ 를 만족하므로 다음과 같다.

$$
\begin{aligned}
Pr[u \rightarrow v\text{: 경로 길이 1}] &= m \cdot Pr[\{u,v\} \in E] \\
&= m \cdot \frac{2}{n^2} \\
&= \frac{2 \cdot m}{n} \cdot \frac{1}{n} \\
&\leq c^{-1} \cdot \frac{1}{n}
\end{aligned}
$$

따라서, ℓ=1 인 경우에 대해서 관계식을 만족한다.

임의의 두 점 u, v에 대해 연결 요소 경로의 길이가 $\ell - 1$이 발생할 확률은

$Pr[\exists u \to v$: 경로 길이 $\ell - 1] \leq c^{-(\ell-1)} \cdot \frac{1}{n}$ 을 만족한다고 가정한다.

$$\begin{aligned} Pr[\exists u \to v \text{경로 길이} \ell] &= Pr[\bigcup_k \exists u \to k \text{ 경로 길이} \ell - 1 \cap \{k, v\} \in E] \\ &\leq \sum_k Pr[\exists u \to k \text{ 경로 길이 } \ell - 1 \cap \{k, v\} \in E] \\ &\leq \sum_k c^{-(\ell-1)} \cdot \frac{1}{n} \cdot c^{-1} \cdot \frac{1}{n} \\ &= \sum_k c^{-\ell} \cdot \frac{1}{n^2} \ (k \leq n) \\ &= c^{-\ell} \cdot \frac{1}{n} \end{aligned}$$

따라서, 임의의 두 점 u, v에 대해 연결 요소 경로의 길이가 ℓ일 확률은 $Pr[\exists u \to v$: 경로 길이 $\ell] \leq c^{-\ell} \cdot \frac{1}{n}$을 만족한다. 이는 수식 (5.2)에서 $Pr[e_i \in L(x_j)] = O(\frac{1}{r})$ 이므로 제안 기법의 동적 재배치 과정은 O(1)의 시간 복잡도를 가진다.

$$\begin{aligned} E[L(x_j)] &= \sum_i^m Pr[e_i \in L(x_j)] \\ &= m \cdot Pr[e_i \in L(x_j)] \\ &\leq m \cdot \sum_{\ell \geq 1} c^{-\ell} \cdot \frac{1}{n} \\ &\leq \frac{m}{n} \cdot \sum_{\ell \geq 1} c^{-\ell} \\ &= O(\frac{m}{n}) \end{aligned} \tag{5.4}$$

□

따라서, 다양한 제안 워크 로드 모델 상에서 발생하는 삽입 연산은 메인 해시 테이블에서 2번의 해시 값으로 계산 되므로 O(1)이고 지연된 동적 재배치 과정의 시간 복잡도 역시 O(1)이다. 이는 기존 기법과 다르게 유휴 시간에 메인 해시 테이블에 배치함으로써 최악의 상황에서도 상수 시간의 삽입 속도를 가지도록

유지하고, 자주 사용되는 데이터들을 우선적으로 이동시킴으로서 검색 시 참조의 지역성 효과를 추가적으로 얻을 수 있다.

검색 (lookup) 연산은 $h_1(x)$와 $h_2(x)$의 값을 계산하고 먼저 해당 위치를 두 개의 해시 테이블에서 데이터를 찾는다. 해시 테이블에서 2개의 해당 위치에 빈 슬롯이 하나라도 있는 경우, 연결 리스트의 탐색은 불필요하다. 따라서, 해당 키값에 해당하는 데이터가 존재하면 해당 데이터 값을 리턴하고 존재하지 않으면 거짓을 리턴한다. 반면, 해시 테이블 2개의 위치에 모두 데이터 노드가 할당되어 있는 경우에 해당 키값에 해당하는 데이터가 존재하면 해당 데이터 값을 리턴하지만 존재하지 않는 경우, 연결 리스트에서 데이터를 찾아야 한다. 해시 값 계산의 연산 비용은 가정에 의해 상수 시간 $\theta(1)$이므로 검색(lookup) 연산은 2번의 해싱 위치의 방문과 연결 리스트 순회 횟수에 비례한다. 수식 (5.4)에서 메인 해시 테이블에서 평균적인 연결 요소 길이 $E[L(x_j)]$는 $(\sum_{\ell>1} c^{-\ell}) \cdot \frac{m}{n}$의 상한을 갖는 것을 보였다. 이는 해시 테이블에 m개의 데이터 노드가 있는 경우 각 슬롯당 연결 리스트에 존재하는 데이터 노드의 수를 의미한다. 따라서, 검색 연산의 경우 수식 (5.5)와 같이 정리될 수 있다.

$$\begin{aligned} E[\tau_{Lookup}(x)] &\leq 2 + (\sum_{\ell>1} c^{-\ell}) \cdot \frac{m}{n} \\ &= O(\frac{m}{n}) \end{aligned} \tag{5.5}$$

이것은 임의의 상수 $c \geq 1$에 대해 $m \leq \frac{n}{2c}$를 만족할 때, 검색 연산은 평균적으로 상수 시간에 수행됨을 의미한다.

삭제와 갱신 연산은 해시 테이블에서 위에서 설명된 검색 연산 수행 후 해당 데이터를 삭제 혹은 갱신한다. 메인 해시 테이블에 있는 데이터 노드의 삭제 및 갱신을 상수 시간으로 가정하면, 삭제와 갱신 연산은 검색 연산의 시간 복잡도와 같으므로 O(1) 이다.

제 6 장 데이터 복구 기법

이 장에서는 결함허용을 위해 메인 메모리 데이터를 백업 서버에 분산하여 저장하고 빠르게 복구하는 기법을 제안한다. 또한 분산 노드들의 결함율을 고려하여 사용자 요청의 예상 응답 시간을 예측하는 모델을 제안하고 다수의 백업 서버를 사용했을 때 복구 시간이 단축됨을 확인한다.

6.1 데이터 백업

제안 기법이 적용된 시스템 모델은 메모리상에 데이터의 사본을 유지하며, 데이터 원본을 저장하는 프로세스를 데이터 서버, 데이터 사본을 저장하는 프로세스를 백업 서버로 지칭한다. 이들 데이터 서버와 백업 서버는 1:N 대응이며 이를 마스터-슬레이브 관계를 맺는다. 백업 서버는 자신의 마스터인 데이터 서버에서 데이터의 사본을 복제하며, 복제한 데이터를 디스크에 백업하는 역할을 수행한다. 데이터 서버에 오류가 발생할 경우 백업 시스템은 백업 서버로부터 데이터를 복구한다.

기존 기법은 데이터를 디스크에 백업하므로 메모리와 디스크의 처리 속도 차이로 인해 복구 시간이 느리다는 문제점이 있다. 또한 데이터 크기가 커질수록 복구 시간이 길어지게 된다. 하지만 제안 기법을 적용하고자 하는 실시간 금융 시스템은 응용의 시스템의 복구 시간을 단축시키는 것이 서비스 품질 보장을 위해 매우 중요하므로 디스크 기반의 백업 기법은 적합하지 않다. 따라서 본 논문에서는 분산 메모리의 데이터 사본을 백업 서버의 메모리에 저장하며, 데이터를 다수의 백업 서버로 분산 적재함으로써 복구 속도를 향상시키는 기법을 제안한다. 제안 기법은 데이터 서버의 데이터를 분할하여 다수의 백업 서버의 메모리에 나누어 보관하며, 오류가 발생했을 시 다수 백업 서버로부터 동시에 데이터를 전송받아 복구한다. 따라서 데이터 크기가 증가하더라도 단일 백업 서버를 사용

하는 시스템에 비해 빠른 속도로 복구가 가능하다. 각 데이터 서버로부터 연결된 백업 서버로의 데이터 분배에는 해시 함수가 사용된다.

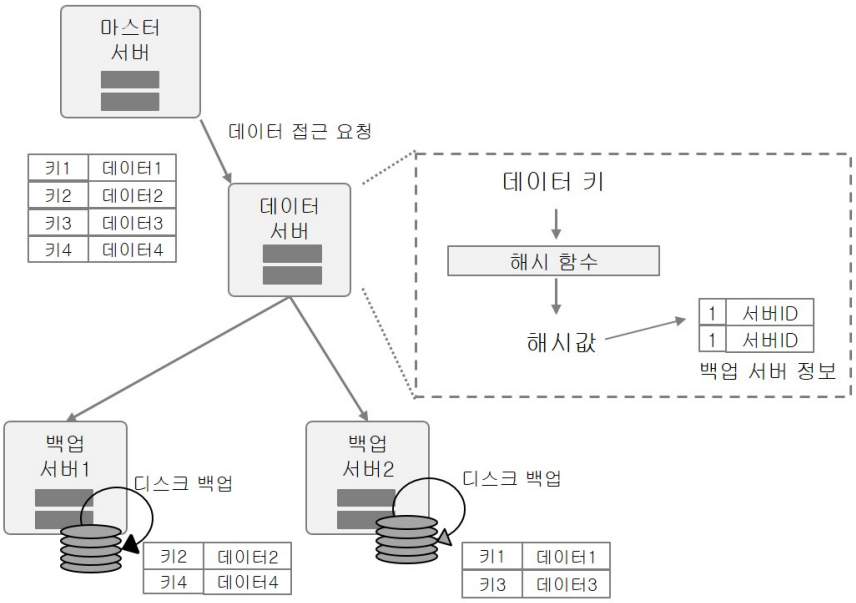

그림 6.1 데이터 백업 구조

6.2 데이터 복구

그림 6.2는 제안 기법의 클러스터 구성으로, 2개의 사본을 유지하는 클러스터의 예시이다. 각 데이터 서버는 자신에게 할당된 데이터를 균등 분할하여 자신의 두 백업 서버에 분배하며, 백업 서버는 자신에게 할당된 데이터 사본을 비동기적으로 디스크에 백업한다. 또한 데이터 서버와 그 서버에 연결된 백업 서버들을 각각 물리적으로 다른 기기에 동작시킴으로써, 데이터 서버에 오류가 발생했을 시 메모리에서 메모리로 데이터를 전송하여 빠른 시간 안에 데이터를 복구할 수 있다.

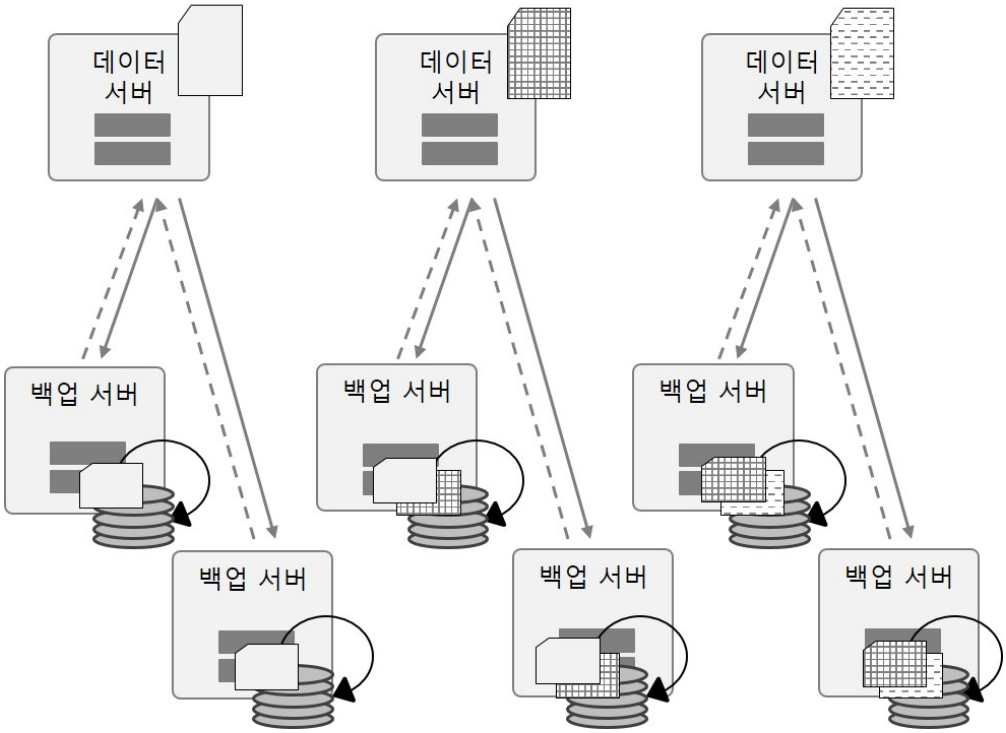

그림 6.2 백업 및 복구를 위한 클러스터 설계

6.3 예상 처리 시간 모델 기반 성능 분석

이 절에서는 분산 메모리 시스템에서 사용자 요청 처리에 필요한 예상 시간을 기반으로 제안하는 복구 기법의 성능을 분석한다. 마스터 서버에서 요청 후 데이터 서버에서 처리가 되기 전까지 오류가 발생하지 않는 경우 분산 메모리의 처리 시간은 t와 같다. 이 처리 시간은 분산 메모리 환경 내 통신 속도에 의해 결정된다. 그러나, 데이터 요청에 대한 서비스가 마치기 전에 데이터 서버에 오류가 발생하면 서비스를 실패하게 된다. 이런 경우, 정상적인 서비스를 위해서는 연결된 백업 서버에 의한 복구 시간(r)과 거치게 되어 마스터 서버에서 재전송하는 시간을 거치게 된다. 메모리 기반 시스템은 높은 가용성을 요구하는 응용에서 사용되므로 복구 시간을 줄이기 위해 m 개의 백업 서버가 연결되어 있다고 가정한다. 또한 전송 실패가 반복적으로 발생할 수 있다고 가정하면, 사용자 요청에 대한 예상

표 6.1 예상 처리 시간 모델 위한 기호 및 함수

기호 및 함수	설명
t	오류가 없을 때, 마스터 서버 요청 후 데이터 서버에서 처리되기까지 시간
$E[t]$	마스터 서버 요청 후 데이터 서버에서 처리될 때까지 예상 처리 시간
r	데이터 서버가 단일 백업 서버에 의해 복구되는 시간
m	하나의 데이터 서버에 연결된 백업 서버의 수
X	서비스 중 전송 실패가 발생할 때까지 걸린 시간을 나타내는 확률 변수
$f_X(x)$	X의 확률 밀도 함수
λ	데이터 서버의 오류 발생 빈도

처리 시간은 재귀적으로 표현해야 한다. X를 마스터 서버에서 요청한 시점을 기준으로 서비스 중에 전송 실패가 발생할 때까지 걸린 시간을 나타내는 확률 변수로 두고. $f_X(x)$를 X의 확률 밀도 함수(probability density function)라고 하자. 시스템 모델을 위한 기호 및 함수는 표 6.1과 같다.

사용자가 데이터 처리를 요청한 시간을 기준으로 반복적인 오류가 발생할 수 있다는 가정을 하는 경우, 예상 처리 시간은 정리 2와 같이 유도할 수 있다.

정리 2 *m개의 백업 서버가 연결된 데이터 서버의 예상 처리 시간 $E[t]$는 다음과 같다.*

$$E[t] = \frac{\int_0^t (\frac{\tau}{m}) f_X(x)dx + t}{\left\{1 - \int_0^t f_X(x)dx\right\}}$$

증명 $X = x$인 경우, 분산 메모리에 요청이 발생한 시점에서 백업 복구 시간을 고려한 예상 응답 시간의 조건부 확률 과정은 다음과 같이 주어진다.

$$E[t] = \begin{cases} t + \frac{\tau}{m} + E[t] & \text{if } x < t \\ \\ t & \text{otherwise} \end{cases} \tag{6.1}$$

즉, 마스터 서버에서 질의를 전달하여 데이터 서버에서 서비스를 제공할 때까지 오류가 발생하지 않는 경우 처리 시간은 t 인 반면, 오류가 발생하여 x 가 t 보다 작은 경우 추가적인 복구 과정과 재전송 과정의 시간을 반복한다. 이 식을 조건부 확률의 전체 기대값의 정리 (law of total expectation) 를 이용해 정리하면 다음 수식을 유도할 수 있다[58].

$$E[t] = \int_0^t (t + \frac{\tau}{m} + E[t]) f_X(x) dx + \int_t^\infty t f_X(x) dx \tag{6.2}$$

여기서 $\int_0^\infty f_X(x)dx = 1$ 이므로, 다음과 같은 식을 얻을 수 있다.

$$\begin{aligned} E[t] &= \int_0^t (t + \frac{\tau}{m} + E[t]) f_X(x) dx + t \int_t^\infty f_X(x) dx \\ &= \int_0^t (t + \frac{\tau}{m} + E[t]) f_X(x) dx + t \Big\{ \int_0^\infty f_X(x) dx - \int_0^t f_X(x) dx \Big\} \\ &= \int_0^t (t + \frac{\tau}{m} + E[t]) f_X(x) dx + t \Big\{ 1 - \int_0^t f_X(x) dx \Big\} \\ &= \int_0^t (\frac{\tau}{m} + E[t]) f_X(x) dx + t \end{aligned}$$

이 식은 다음과 같이 정리된다.

$$E[t] \Big\{ 1 - \int_0^t f_X(x) dx \Big\} = \int_0^t (\frac{\tau}{m}) f_X(x) dx + t$$

정리하면 마스터 서버로부터 요청된 데이터 서버의 예상 처리 시간 $E[t]$ 는 다음과

같다.

$$E[t] = \frac{\int_0^t (\frac{\tau}{m}) f_X(x)dx + t}{\left\{1 - \int_0^t f_X(x)dx\right\}}$$

□

그리고 데이터 서버의 오류가 발생 빈도 λ를 가지는 포아송 과정을 따른다고 가정하면, 마스터 서버로 부터 데이터 서버의 처리 시간은 정리 3과 같이 유도할 수 있다.

정리 3 *데이터 서버의 오류 발생 빈도가 λ의 비율을 가지는 포아송 과정을 따르고 m 개의 백업 서버를 유지할 때, 예상 처리 시간 $E[t]$ 는 다음과 같이 주어진다. 단, $\lambda > 0$ 이다.*

$$E[t] = (\frac{\lambda \cdot \tau}{m} + t)e^{\lambda t} - \frac{\lambda \cdot \tau}{m}$$

증명 데이터 서버가 λ의 비율을 가지는 포아송 과정으로 오류가 발생하므로 마스터 서버로 부터 요청된 질의가 데이터 서버로 부터 실패할 확률 분포는 $\lambda e^{-\lambda x}$ 으로 주어진다. 정리 2에서 $f_X(x)$을 $\lambda e^{-\lambda x}$ 으로 대치하면 다음과 같은 수식을 유도할 수 있다.

$$E[t] = \frac{\int_0^t (\frac{\tau}{m}) \lambda e^{-\lambda x} dx + t}{\left\{1 - \int_0^t \lambda e^{-\lambda x} dx\right\}}$$

여기서 $\int_0^t \lambda e^{-\lambda x} dx = 1 - e^{-\lambda t}$ 이므로,

$$
\begin{aligned}
E[t] &= \frac{\int_0^t (\frac{\tau}{m})\lambda e^{-\lambda x} dx + t}{\left\{1 - (1 - e^{-\lambda t})\right\}} \\
&= \frac{\int_0^t (\frac{\tau}{m})\lambda e^{-\lambda x} dx + t}{e^{-\lambda t}} \\
&= \frac{\frac{\lambda \cdot \tau}{m} \int_0^t e^{-\lambda x} dx + t}{e^{-\lambda t}} \\
&= \frac{\frac{\lambda \cdot \tau}{m}(1 - e^{-\lambda t}) + t}{e^{-\lambda t}} \\
&= \frac{\lambda \cdot \tau}{m}(e^{\lambda t} - 1) + te^{\lambda t} \\
&= (\frac{\lambda \cdot \tau}{m} + t)e^{\lambda t} - \frac{\lambda \cdot \tau}{m}
\end{aligned}
$$

□

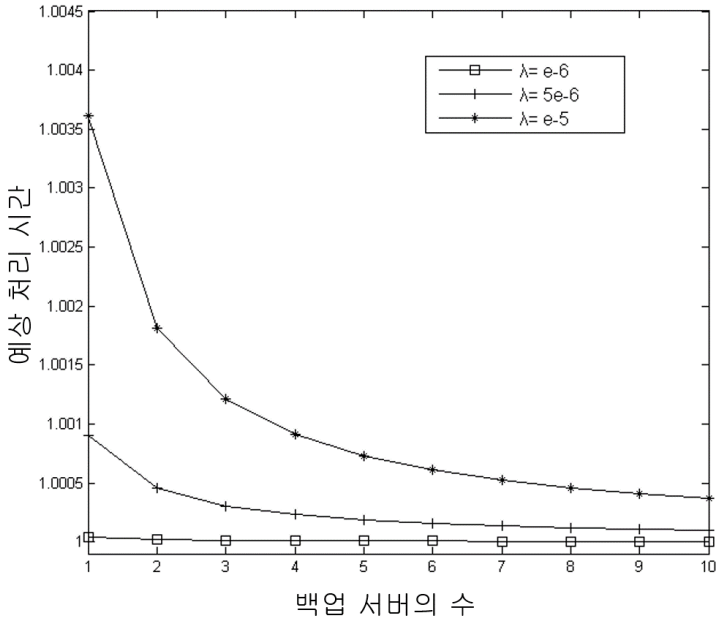

그림 6.3 예상 처리 시간

분산 메모리 환경에서 요청한 키값에 해당하는 데이터 서비스를 위한 예상

처리 시간은 데이터 서버의 오류 빈도에 지수적으로 증가한다. 하지만 데이터 서버에 대한 백업 서버들의 수가 많을수록 처리 시간은 백업 서버의 수에 반비례한다. 이는 백업 서버의 수가 증가할수록 처리 시간에 어떻게 영향을 주는지를 수학적으로 분석 가능함을 의미한다.

그림 6.3은 분산 메모리 환경에서 백업 서버 구성에 따른 정리 3의 예상 처리 시간을 보여준다. 데이터 서버에 연결된 백업 서버의 수(m)가 증가할수록 예상 처리 시간은 감소하는 것을 볼 수 있다. 이는 오류 발생 시 다수의 백업 서버에서 동시에 데이터 서버로 백업 데이터를 전송하기 때문이다. 특히 오류 발생율 λ가 $10^6, 5 \cdot 10^6, 10^5$ 인 환경에서 백업 서버 수가 각각 $M > 1, 4, 8$ 일 때 예상 처리 시간의 효과는 거의 없었다 ($<0.01\%$). 이것은 제안하는 예상 처리 시간 모델이 오류 발생율이 다른 메모리 기반 시스템 환경에서 효과적인 백업 서버 구성에 사용될 수 있음을 의미한다.

제 7 장 성능 평가

본 장에서는 제안 기법을 구현하고 실험을 통해 측정한 결과를 기술한다. 분산 메모리 시스템에서 효율적인 데이터 관리를 위해 제안한 해시 기법과 복구 기법이 데이터 처리의 효율성 복구 지연 시간 등에서 어느 정도의 성능을 보이고 있는지 실험하였다. 또한 실제 금융 거래 시스템의 워크로드를 사용하여 제안 기법의 성능을 측정하였다.

7.1 구현 및 실험 환경

리눅스 기반의 분산 메모리 관리 기법인 Redis[14]를 기반으로 분산 메모리 시스템을 구축하고 제안 알고리즘을 적용하였다. 그림 7.1은 제안 기법의 실험 환경을 나타낸다. 인피니밴드 네트워크로 연결된 3대의 서버 PC가 클러스터를 구성한다. 사용자 요청 발생기는 실제 금융 거래 시스템에서 사용되는 데이터를 수집한 것을 바탕으로 사용자의 요청을 발생시키며, 이를 받은 응용 프로그램에서 분산 메모리 시스템의 API를 이용하여 마스터 서버에 데이터에 대한 접근을 요청한다. 마스터 서버는 각 서버 PC마다 존재하며, 접근할 데이터 서버를 선택하여 데이터 접근 요청을 전달한다. 각 데이터 서버는 본 논문에서 제안한 해시 기법을 사용하여 데이터를 관리하며, 각각 2개의 백업 서버를 물리적으로 다른 장치에 유지 관리한다.

성능 평가에서 사용된 실험 환경은 다음과 같다.

표 7.1 실험 환경

CPU	Intel Core 2 Quad Q9400 2.66GHz
메모리	4GB
운영체제	centOS6.4
커널 버전	2.6.32-358.6.1

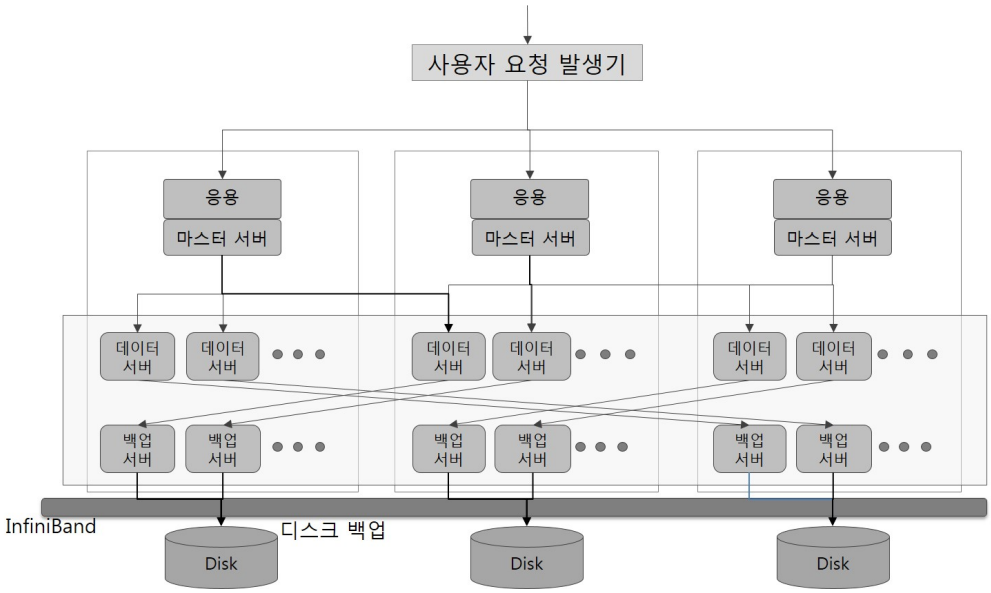

그림 7.1 실험 환경

7.2 제안하는 해시 기법 성능 평가

그림 7.2, 7.3, 7.4는 체인 해시 기법과 제안 해시 기법을 사용하여, 테이블 슬롯의 40%가 사용된 상태에서 검색 작업에 걸린 시간을 측정한 것이다. 각각 10만에서 100만 건의 성공적인 검색, 실패한 검색, 10% 참조 지역성이 있는 검색을 수행하였다. 실험 결과 약 90%의 키가 메인 해시 테이블에 저장됨을 확인하였다. 제안 해시 기법은 체인 해시 기법에 비해 성공적인 검색의 경우 평균 7.4%, 최대 15%, 10%의 지역성이 있는 경우 평균 13%, 최대 24% 적은 시간이 소요되었다. 반면 실패한 검색의 경우 평균 4.6%, 최대 12% 느린 속도를 보였는데, 검색에 실패할 경우 메인 해시 테이블 슬롯과 보조 해시 테이블의 리스트를 모두 검색해야하기 때문이다. 하지만 검색이 실패하는 경우에도 제안 기법에서는 체인 해시 테이블의 리스트가 대부분 1개의 데이터만을 저장하고 있다(표 7.2). 최악의 경우 검색 시간은 제안 해시 기법이 더 짧음을 예상할 수 있다.

표 7.2 해시 테이블 참조 횟수

해시 기법	평균	최악의 경우
체인 해시 기법	1.17	7
제안 해시 기법	1.13	3

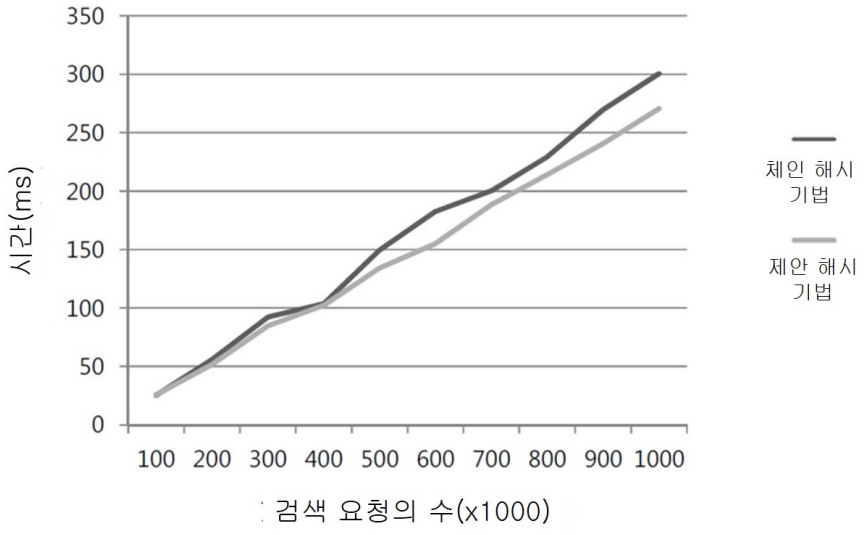

그림 7.2 데이터를 찾을 경우 검색 시간

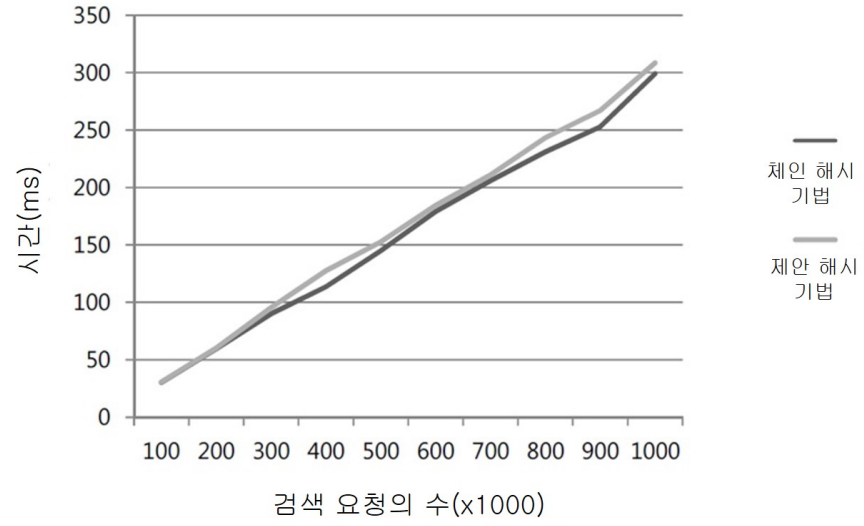

그림 7.3 데이터를 못 찾을 경우 검색 시간

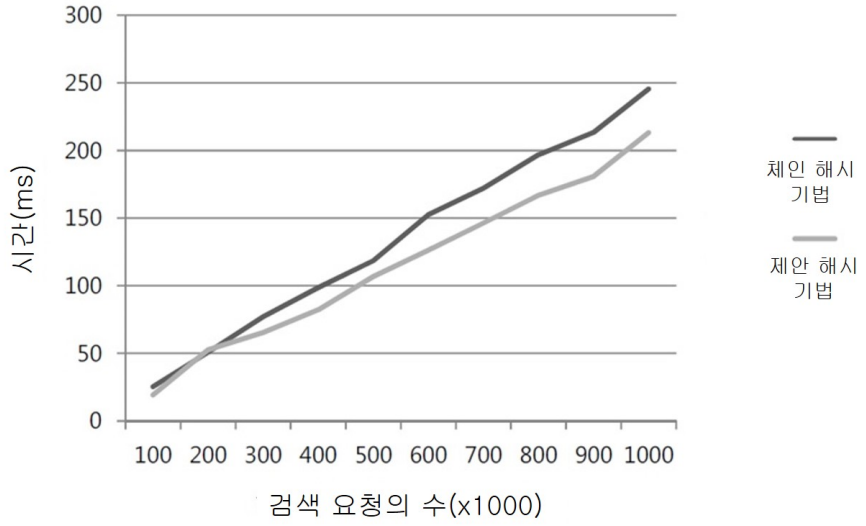

그림 7.4 10% 지역성이 있는 경우 검색 시간

4.2.4절에서 설명한 바와 같이 제안 해시 기법에 블룸 필터를 사용했을 때와 보조 해시 테이블에서 참조된 데이터를 리스트 헤드로 이동시켰을 때 검색(Lookup) 연산의 성능 변화를 확인하였다. 그림 7.5는 블룸 필터를 적용한 결과로서, 가로축은 해시 테이블의 부하율(Load Factor)을 나타내며 세로축은 검색 연산의 초당 처리율을 나타낸다. 부하율에 따라 데이터를 먼저 삽입한 후 검색 연산의 초당 처리율을 측정하였다. 부하율이 높아지면 충돌이 발생할 확률이 높아지므로 보조 해시 테이블에 저장된 데이터의 비율이 높아지고, 따라서 보조 해시 테이블의 탐색 비용이 커짐을 의미한다. 각 키는 100바이트의 문자열을 사용하였으며, 그래프의 각 데이터 값은 20회 반복한 결과의 평균이다.

실험 결과 처리율이 부하율 10%일 때 약 2%, 70%일 때 약 12% 개선됨을 확인하였다. 부하율이 증가할수록 성능의 차이가 커짐을 알 수 있다. 블룸 필터를 사용하면 필터가 거짓을 리턴했을 때 해시 테이블 내에 접근하지 않고 바로 리턴하는데, 충분히 큰 비트맵을 사용하면 정확도를 높일 수 있다. 부하율이 작을 때는 큰 차이를 보이지 않는데, 블룸 필터가 다수의 해시 함수를 사용하므로 얻을 수 있는 이득과 비용이 상쇄되기 때문이다. 하지만 부하율이 클수록 블룸 필터를 사용하지 않은 경우 보조 테이블에 접근하여 데이터 키값을 비교, 검색하는

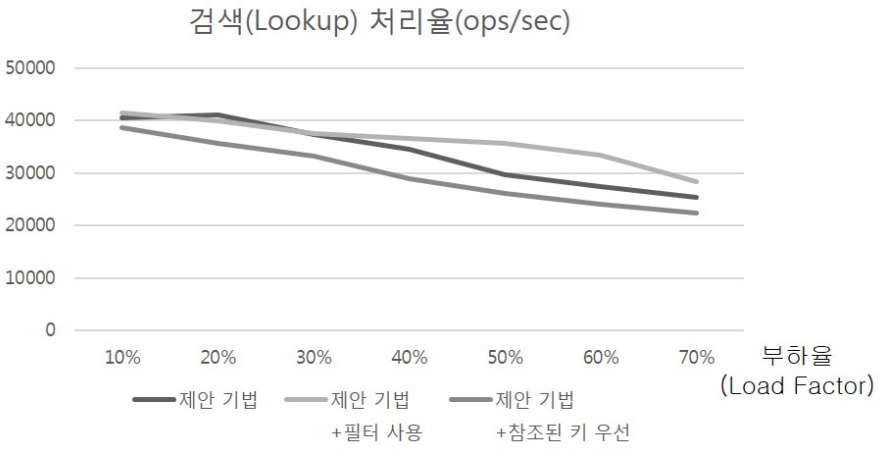

그림 7.5 블룸 필터 사용 시 검색 연산 처리율

비용이 커지게 되므로 필터를 사용하여 테이블에 접근하지 않을 때에 비해 기회 비용이 증가한다.

또한 그림 7.5는 블룸 필터 적용 결과와 함께 보조 해시 테이블에서 두 번 이상 참조된 데이터를 리스트 헤드로 이동시켰을 때 처리율을 비교한 결과를 나타내고 있다. 실험 결과 이 기법을 적용하지 않았을 시에 비해 처리율이 최대 12%까지 떨어짐을 확인하였다. 이는 보조 해시 테이블의 리스트 탐색이 각 노드당 두 번의 포인터 참조가 필요한 데 비해, 데이터를 리스트 헤드로 이동시키기 위해서는 총 네 번의 포인터 참조가 필요해지기 때문이다. 따라서 이 참조 우선 정책이 효과적이기 위해서는 한 번 참조된 데이터가 다시 참조될 확률이 높아야 하며, 해당 데이터가 리스트 내에서 세 번째 이상에 있을 때만 이익이 발생한다. 대부분의 경우 이익보다 비용이 더 큰 상황이 발생하므로 이 정책을 사용하지 않았을 때보다 성능이 떨어짐을 알 수 있다. 데이터가 다시 참조될 확률이 얼마나 되는가에 따라 결과가 달라지므로 향후에는 보다 다양한 워크로드를 대상으로 실험해볼 필요성이 있다.

7.3 제안하는 복구 기법 성능 평가

그림 7.6은 제안한 복구 기법의 성능 평가를 위해 데이터의 숫자가 증가함에 따른 복구 시간을 측정한 결과이다. 단일 백업 서버를 사용한 경우에 비해 두 개의 백업 서버를 사용한 경우 복구 시간이 평균 51.4% 감소함을 확인하였다.

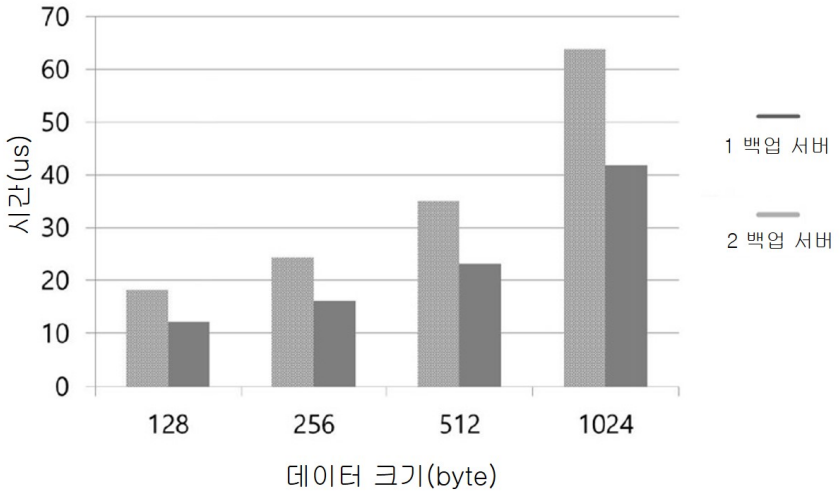

그림 7.6 데이터 복구 시간

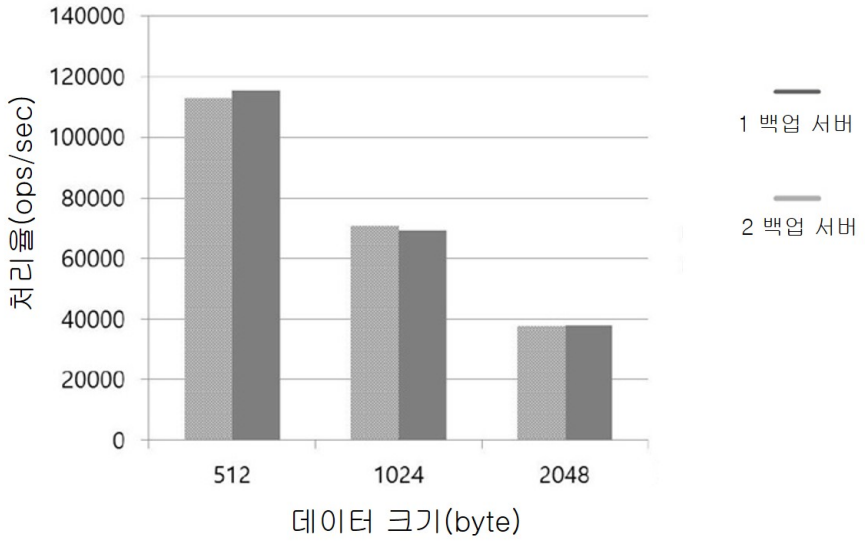

그림 7.7 데이터 복구 오버헤드

그림 7.7은 제안한 복구 기법의 오버헤드를 측정하기 위해 데이터의 숫자가 증가함에 따른 마스터 서버의 처리율을 측정한 결과이다. 단일 백업 서버를 사용한 경우와 두 개의 백업 서버를 사용한 경우 복구 시간이 거의 동일함을 확인할 수 있다.

7.4 분산 메모리 시스템 성능 평가

제안 기법을 실제 금융 시스템에 통합하여, 금융 애플리케이션을 사용했을 때 분산 메모리 시스템의 처리 성능을 측정하였다. 실험은 인피니밴드 기반의 3개 서버로 구성된 클러스터에서 시행되었다. 표 7.3은 3만 건의 주문을 발생시켰을 때의 응용 서버에서 측정한 소요 시간을 나타낸다. 하나의 주문은 7개 9의 연산으로 이루어져 있으며, 초당 1만 건 이상의 주문을 처리할 수 있음을 확인하였다. 표 7.4는 위 실험 결과의 주문별 소요 시간을 처리과정 별로 상세히 나타낸 것이다. 각 주문은 데이터베이스로의 다수 읽기 및 쓰기 요청과 논리적인 처리 부분으로 구성된다. 실험 결과 읽기 요청의 지연시간은 70 us 이하였으며 쓰기(삽입 및 수정) 요청의 지연 시간은 90 us 이하임을 확인하였다.

표 7.3 분산 메모리 시스템 성능 평가

반복 회수	회차별 주문 건수	평균 소요 시간(us)	초당 주문 처리량
20	30,000	2,619,915	11,469

표 7.4 단일 주문의 주문 단계별 소요 시간

처리 단계	요청 분류	쓰레드1	쓰레드2	쓰레드3
주문코드	검색	56	55	56
사용자계좌	검색	67	67	66
코드-계좌	검색	64	61	65
로직	-	2	2	2
포트폴리오	검색	51	49	51
로직	-	0	0	0
주문정보	삽입	88	86	88
로직	-	0	0	0
사용자계좌	갱신	8	87	87
코드-계좌	갱신	76	74	76

제 8 장 결론 및 향후 연구 방향

8.1 결론

최근 다중 서버 메모리 기반 시스템들은 응용 범위가 넓어지고 복잡한 기능이 요구되면서 더 높은 수준의 성능이 필요하게 되었다. 메모리 기반 시스템들은 일반적인 대규모 데이터 관리뿐 아니라 적절한 서비스 품질을 요구하는 웹서비스, 빠른 성능이 요구되는 금융 거래 서비스 등 다양한 특징을 가진 응용들에 사용되고 있다. 이에 따라 메모리 기반 시스템은 데이터 정합성, 확장성, 처리 성능 등 다양한 요구 조건을 충족시켜야 하게 되었다. 최근 메모리 기반 시스템에 대한 연구가 활발해지고, 처리해야 하는 데이터 크기가 커짐에 따라 시스템의 성능을 높이기 위한 메모리 데이터 관리의 중요성이 대두되고 있다. 메모리의 데이터를 적절히 분배하고 저장 및 관리하는 방법에 따라 시스템의 성능이 크게 좌우되게 된다. 메모리 기반 시스템의 데이터 관리 기법은 다양한 기업 및 연구소에서 활발히 연구되고 있으나, 그 성능이나 안정성에 있어서 아직까지 연구 개발의 여지가 많이 남아 있는 실정이다. 특히 분산 메모리 데이터 접근 시 처리 속도에 중요한 역할을 하는 해시 알고리즘에 있어서 기존의 관리 기법들은 읽기 중심의 워크로드에 특화되어 있어 점차 다양해지는 응용들의 요구사항을 충족시키지 못하고 있다. 본 논문에서는 분산 메모리의 데이터를 효율적으로 저장하고, 읽기뿐만 아니라 쓰기 중심의 워크로드에서도 효율적으로 데이터를 관리할 수 있는 기법을 제안하였다. 기존 쿠쿠 해시 기법은 읽기 요청을 처리하는 데는 적합하지만, 쓰기 요청이 많아졌을 때는 요청의 응답 시간을 보장할 수 없는 문제점이 존재한다. 따라서 제안 기법에서는 쿠쿠 해시 알고리즘을 사용하는 메인 해시 테이

블과 연결 리스트를 저장하는 보조 해시 테이블을 사용하여, 쓰기 요청된 데이터를 보조 해시 테이블에 먼저 보관하도록 하였다. 시간 복잡도 기반 성능 분석을 통해 검색, 삽입 연산 등이 상수 시간에 처리됨을 보였다. 또한 분산 메모리 데이터의 결함 혀용을 위해 각 데이터 서버가 다수 백업 서버의 메모리에 데이터를 분산 백업하고, 복구 시 데이터를 동시에 전송하도록 함으로써 복구 시간을 줄이는 기법을 제안하였다. 제안 복구 기법을 수학적으로 분석하여 성능을 확인하였으며, 제안 해시 기법과 복구 기법을 구현한 후 실제 데이터를 사용해 실험하여 제안 기법이 효과적으로 다양한 워크로드에서 높은 성능을 낼 수 있음을 확인하였다.

8.2 향후 연구 방향

먼저 제안 해시 기법의 성능을 높이기 위해 보조 해시 테이블의 활용 방안과 해시 함수 설계 등을 연구할 계획이다. 또한 사용자 요청의 패턴에 따라 각 해시 테이블의 최적 크기를 동적으로 조절하는 기법을 연구할 계획이다. 마지막으로 최근 다양한 연구가 이루어지고 있는 인피니밴드의 RDMA 기술을 메모리 데이터 관리 기술을 접목하여 응답 시간을 줄이기 위한 연구 또한 유의미할 것이다.

참고문헌

[1] M.K. Gupta, V. Verma, and M.S. Verma, "In-Memory Database Systems – A Paradigm Shift," *International Journal of Engineering Trends and Technology*, vol. 6, no. 6, pp. 333-336, December 2013.

[2] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A Memory-efficient, Highperformance Key-Value Store," in *Proceedings of ACM Symposium on Operating Systems Principles*, Cascais, Portugal, pp. 1-13, October 2011.

[3] B. K. Park, W. Jung and J. Jang, "Integrated Financial Trading System based on Distributed In-Memory Database," in *Proceedings of ACM Conference on Research in Adaptive and Convergent Systems*, Towson, MD, USA, pp. 356-358, October 2014.

[4] R. Cattel, "Scalable SQL and NoSQL Data Stores," *ACM SIGMOD Record.*, vol. 39, no. 4, pp. 12-27, December 2010.

[5] *AMD64 Architecture Programmer's Manual*, vol. 2, AMD Corporation, 2012.

[6] *Intel Itanium Architecture Software Developer's Manual*, vol. 3, Intel, 2010.

[7] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509-516, December 1992.

[8] J. Han, "Survey on NoSQL Database," in *Proceedings of International Conference on Pervasive Computing and Applications*, Republic of South Africa, pp. 363-366, October 2011.

[9] R. Hecht and S. Jablonski, "NoSQL Evaluation," in *Proceedings of International Conference on Cloud and Service Computing*, Hong Kong, China, pp. 336-341, December 2011.

[10] B. Atikoglu, et al., "Workload Analysis of a Large-scale Key-Value Store," in *Proceedings of ACM Joint International Conference on Measurement and Modeling of Computer Systems*, London, UK, pp. 53-64, June 2012.

[11] A. Lakshman and P. Malik, Prashant, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Operating System. Review*, vol. 44, no.2, pp. 35-40, April 2010.

[12] S. Zhang, etl al., "Ensembles of Models for Automated Diagnosis of System Performance Problems," in *Proceedings of International Conference on Dependable Systems and Networks*, pp. 644-653, July 2005.

[13] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads," in *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802-1813, August 2012.

[14] J. L. Carlson, "Redis in Action," *Manning Publications Co.*, ISBN:978-1617290855, 2013.

[15] R. Klophaus, "Riak Core: building distributed applications without shared state," in *Proceedings of ACM SIGPLAN Commercial Users of Functional Programming*, Baltimore, MD, USA, September 2010.

[16] B. Fan, D.G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent Memcache with Dumber Chaching and Smarter Hashing," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, Lombard, IL, USA, pp. 371-384, 2013.

[17] J. Ousterhout, et al., "The case for RAMClouds: scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92-105, January 2010.

[18] D. Ongaro, et al., "Fast Crash Recovery in RAMCloud," in *Proceedings of ACM Symposium on Operating Systems Principles*, Cascais, Portugal, pp. 29-41, October 2011.

[19] R. Geambasu, et al., "Comet: An Active Distributed Key-Value Store," in *Proceedings of USENIX Conference on Operating Systems Design and Implementation*, Vancouver, BC, Canada, October 2010.

[20] H. Plattner and A. Zeier, "In-Memory Data Management: Technology and Applications," *Springer*, ISBN:978-3642295744, 2012.

[21] M. N. Vora, "Hadoop-HBase for Large-scale Data," in *Proceedings of International Conference on Computer Science and Network Technology*, Harbin, China, pp. 601-605, December 2011.

[22] N. Sundaram, et al., "Streaming Similarity Search over One Billion Tweets Using Parallel Locality-sensitive Hashing," in *Proceedings of the VLDB Endowment*, pp. 1930-1941, September 2013.

[23] M. Ekman and P. Stenstrom, "A Robust Main-Memory Compression Scheme," in *Proceedings of International Symposium on Computer Architecture*, Wisconsin, USA, pp. 74-85, June 2005.

[24] K. Banker, "MongoDB in Action," *Manning Publications Co.*, ISBN:978-1935182870, 2011.

[25] G. Decandia, et al., "Dynamo: Amazon's Jighly Available Keyvalue Store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205-220, December 2007.

[26] E. Sciore, "SimpleDB: A Simple Java-based Multiuser Syst for Teaching Database Internals," *SIGCSE Bull.*, vol. 39, no.1, pp. 561-565, March 2007.

[27] F. Chang, et al., "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, vol. 26, no.2, pp. 1-26, June 2008.

[28] C. Tesoriero, "Getting Started with OrientDB," *Packt Publishing*, ISBN:978-1782169956, 2013.

[29] J. Webber, "A Programmatic Introduction to Neo4J," in *Proceedings of Annual Conference on Systems, Programming, and Applications: Software for Humanity*, Tucson, USA, pp. 217-218, July 2012.

[30] H. Zhang, et al., "In-Memory Big Data Management and Processing: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no.7, pp. 1920-1948, July 2015.

[31] Oracle white paper, "A Survey of In-Memory Databases from SAP, Microsoft, IBM and Oracle," *ORACLE*, 2015.

[32] P. Xiang, R. Hou, and Z. Zhou, "Cache and Consistency in NOSQL," in *Proceedings of IEEE International Conference on Computer Science and Information Technology*, Chengdu, China, pp. 117-120, July 2010.

[33] E. Malalla, "Two-way Hashing with Separate Chaining and Linear Probing," *Thesis: School of Computer Science, McGill University*, 2004.

[34] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no.12, pp. 1378-1381, December 2008.

[35] A. Z. Broder, M. Najork, and J. L. Wiener, "Efficient URL Caching for World Wide Web Crawling," in *Proceedings of the 12th International Conference on World Wide Web*, Budapest, Hungary, pp. 679-689, May 2003.

[36] E. Brun, A. Guittet, and F. Gibou, "A local level-set method using a hash table data structure," *Journal of Computational Physics*, vol. 231, no.6, pp. 2528–2536, March 2012.

[37] S. Deodhar, and A. L. Tharp, "Shift Hashing for Memory-Constrained Applications," in *Proceedings of IEEE International Computer Software and Applications Conference*, Seattle, USA, pp. 531-536, July 2009.

[38] B. Christel, et al., “Probabilistic Methods in State Space Analysis,” *Validation of Stochastic Systems*, vol. 2925, no.1, pp. 339-383, June 2004.

[39] H. Gao, J. F. Groote, and W. H. Hesselink, “Lock-free dynamic hash tables with open addressing ,” *Distributed Computing*, vol. 18, no.1, pp. 21-42, July 2005.

[40] M. L. Fredman, J. Komlos, and E. Szemeredi, ”Storing a Sparse Table with 0(1) Worst Case Access Time,” *Journal of ACM*, vol. 31, no. 3, pp. 538-544, July 1984.

[41] Hash Table Performance Tests, http://preshing.com/20110603/hash-table-performance-tests/

[42] Y. Song, Z. Li, ”New tiling techniques to improve cache temporal locality,” in *Proceedings of ACM Conference on Programming Language Design and Implementation*, vol. 34, no. 5, pp. 215-228, May 1999.

[43] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, ”Fast hash table lookup using extended bloom filter: an aid to network processing ,” ACM SIGCOMM Computer Communication Review, vol. 35, no. 4, pp. 181-192, October 2005.

[44] B. H. Bloom, ”Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[45] J. Liu, J. Wu, and D. K. Panda, ”High performance RDMA-based MPI implementation over Infiniband,” *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167-198, June 2004.

[46] J. Liu, A. Vishnu, D. K. Panda, ”Building Multirail Infiniband Clusters: MPI-Level Design and Performance Evaluation,” in *Proceedings of ACM/IEEE Cconference on Supercomputing*, Pittsburgh, PA, USA, pp. 33-45, November 2004.

[47] R. Pagh and F. F. Rodler, ”Cuckoo Hashing,” *Journal of Algorithms*, vol. 51, no. 2, pp. 122-144, May 2004.

[48] C. Mitchell, J. Li, and Y. Geng, ”Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store,” in *Proceedings of USENIX Annual Technical Conference*, San Jose, CA, USA, pp. 1-12, June 2013.

[49] A. Kaila, M., Kaminsky, and D. G. Andersen, ”Using RDMA Efficiently for Key-Value Services,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 295-306, October 2014.

[50] S. Reinemo, T. Skeie, T. Sodring, and O. Lysne, "An Overview of QoS Capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet," *IEEE Communication Magazine*, pp. 32-38, July 2006.

[51] B. Cully, et al., "Remus: High Availability via Asynchronous Virtual Machine Replication," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, CA, USA, pp. 161-174, April 2008.

[52] M. Wiesmann, et al., "Understanding Replication in Databases and Distributed Systems," in *Proceedings of International Conference on Distributed Computing Systems*, Taipei, Taiwan, pp. 464-474, April 2000.

[53] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 23-31, January 1987.

[54] L. Devroye, "Cockoo Hashing: Further Analysis," *Information Processing Letters*, vol. 86, no. 4, pp. 215-219, May 2003.

[55] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56-78, February 1991.

[56] J. Zhao, et al., "Kiln: Closing the Performance Gap Between Systems with and without Persistence Support," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, Canada, pp. 421-432, December 2013.

[57] B.F. Cooper, et al., "Benchmarking cloud Serving Systems with YCSB," in *Proceedings of ACM Symposium on Cloud Computing*, Indianapolis, IN, USA, pp. 143-154, June 2010.

[58] J. L. Doob, "Probability as Measure," *The Annals of Mathematical Statistics*, vol. 12, no. 2, pp. 206-214, June 1941.

[59] J. Jang, Y. Cho, J. Jung and J. Hong, "A Fast and Efficient Key-Value Store using Infiniband RMDA," *Information - An International Interdiciplinary Journal.*, vol. 17, no.9(B), pp. 4505-4514, September 2014.

[60] D. Min, T. Hwang, J. jang, Y. Cho and J. Hong, "An Efficient Backup-Recovery Technique to Process Large Data in Distributed Key-Value Store," in *Proceedings of ACM/SIGAPP Symposium on Applied Computing*, Salamanca, Spain, pp. 356-358, April 2015.

[61] J. Jang, J. Jung, Y. Cho and S. Yoon, "Concurrency Control Scheme for Key-Value Stores based on InfiniBand," in *Proceedings of ACM Conference on Research in Adaptive and Convergent Systems*, Towson, MD, USA, pp. 356-358, October 2014.

[62] J. Jang, Y. Cho, J. Jung and J. Hong, "Enhancing Lookup Performance of Key-Value Stores using Cuckoo Hashing," in *Proceedings of ACM Conference on Research in Adaptive and Convergent Systems*, Montreal, QC, Canada, pp. 487-489, October 2013.

[63] 장준혁 외, "Cuckoo hashing을 이용한 Key-Value Store의 검색 성능 개선," *한국정보과학회 2013년 춘계학술대회*, 여수, 한국, pp. 124-126, 2013년 6월.

[64] 장준혁, 정진만, 김봉재, 조유근, "Key-Value Store의 백업과 복구 기술 연구," *2012 순천 정원엑스포 ICT 합동학술대회*, 순천, 한국, pp. 15-17, 2012년 6월.

[65] 장준혁, 조유근, "인메모리 데이터베이스 기반 금융 거래 시스템," *2015 스마트미디어학회 춘계학술대회*, 서울, 한국, pp. 5-7, 2015년 4월.

Abstract

A Multiple Memory Server Management Scheme for Massive Explosive Data Streams

Joonhyouk Jang
Department of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

Abstract

Memory-based systems store and manage massive volume of data in memory of multiple servers instead of disks. Because of sharp deflation in the price of memory, application of the memory-based system expands in many areas. Memory-based systems show better performance than previous systems based on disk in managing the servers that are required to process large volume of data transfer such as real-time trading servers, big data analysis servers, and enterprise servers.

In memory-based systems, the efficient management of data stored in main memory of multiple servers is the key to increase the quality of service and system reliability. However, the performance of existing management

schemes of data in memory sharply degrades when the characteristics of user requirements vary or the number of servers connected via network increases.

Therefore, in this thesis, we propose an efficient data management scheme for memory-based systems using hashing and data recovery. The proposed hashing scheme uses duplicated hash table for Lookup operation, and uses hash table with linked list for Insert operation. By this, both Lookup operation and Insert operation can be performed in constant time. In addition, we propose a data recovery scheme that stores data in memory of multiple backup servers that simultaneously send data to data server when recovery is required.

In order to verify the performance of the proposed scheme, we stochastically analyze the proposed hashing scheme and it is shown that Insert, Lookup, Update and Delete operations are performed in constant time on average. In addition, we also analyze the expected processing time of the data server using the proposed scheme based on probabilistic modeling. To evaluate the performance of the proposed scheme, we implemented our scheme and tested the implementation on actual data workload. Our experiment shows that the proposed scheme increases the throughput and decreases the recovery time.

Keywords: Memory-based System, Data Management, Heavy Workload, Hash Table, Data Recovery, Performance Analysis

Student Number: 2010-20882